

CSE354: Distributed Computing

Project Title:

Distributed Image Processing System using Cloud Computing

Phase 1: Project Planning and Design

Group(46) :

*Eslam Mostafa Mohamed Haider
Belal Mahmoud Amer Mahmoud
Abdelrahman Mostafa
Zeyad Essam Elsayed*

*20P4797
19P2374
20P6060
20P5728*

Submitted to:

Prof. Ayman M. Bahaa-Eldin

Contents

Phase 1: Project Planning and Design	1
# Project proposal	3
1. Introduction	3
2. Project objectives	3
3. Project Scope	4
4. Project Requirements	4
5. Project Deliverables	5
# Design document	7
6. System Architecture Design	7
7. Project Plan and timelines	10
8. User Stories	13

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.

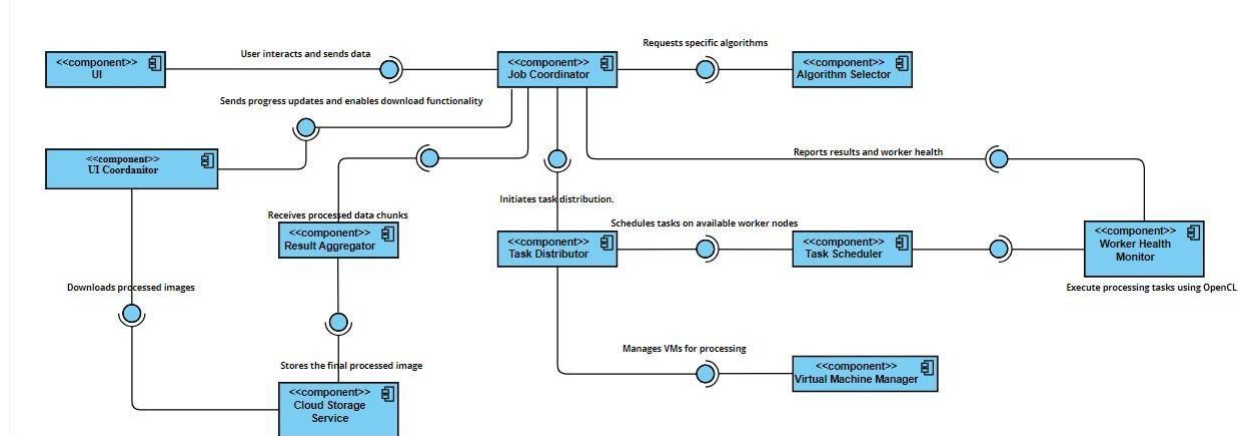
Communication Flow:

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.

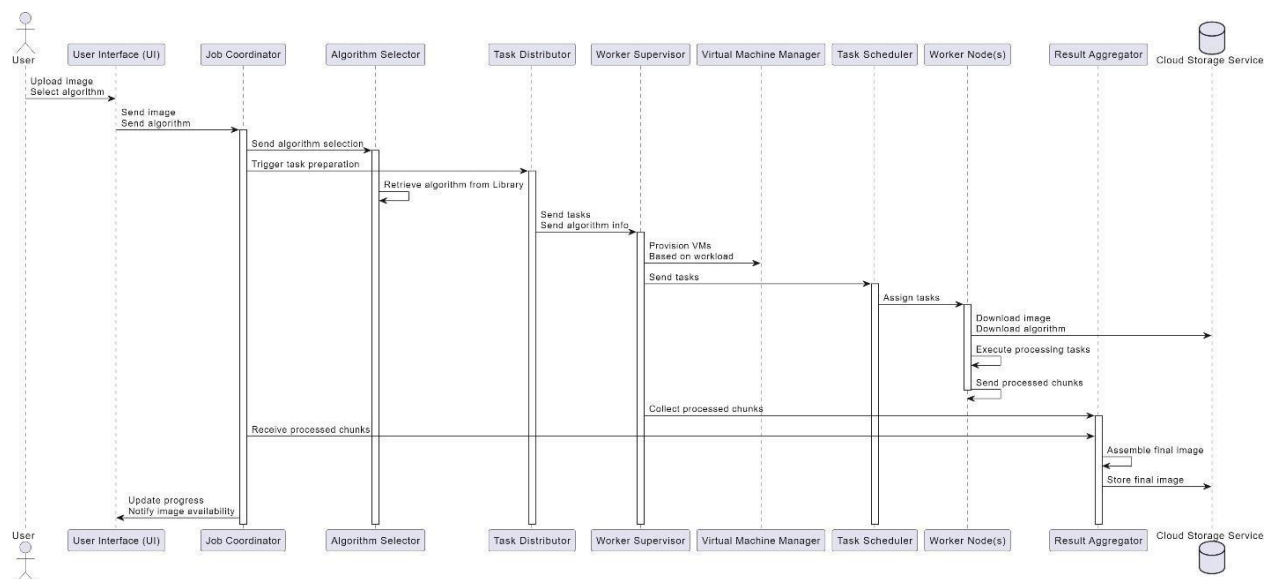
Benefits of this Architecture:

- **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.

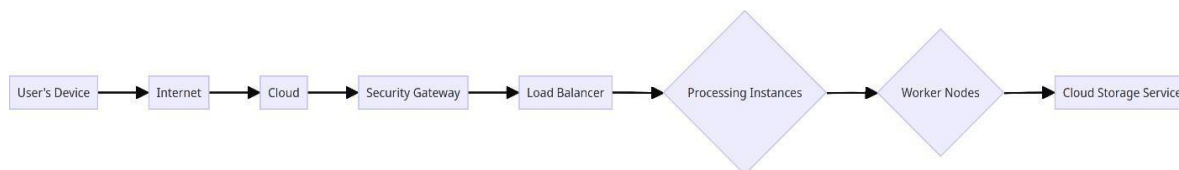
Component Diagram: -



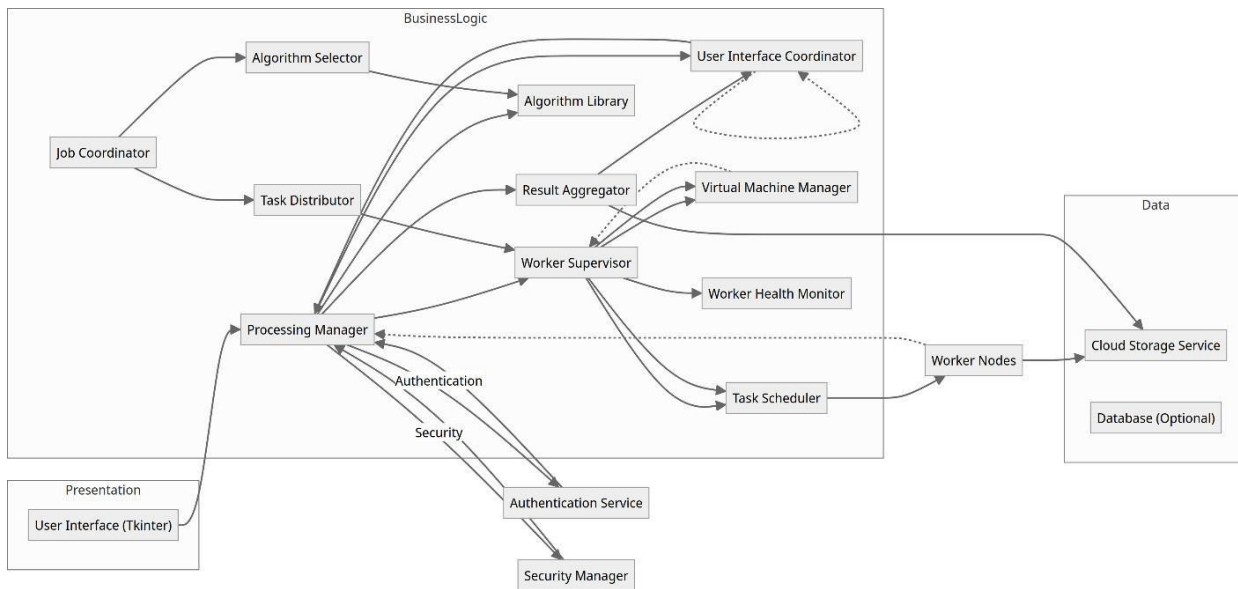
Sequence Diagram: -



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.

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

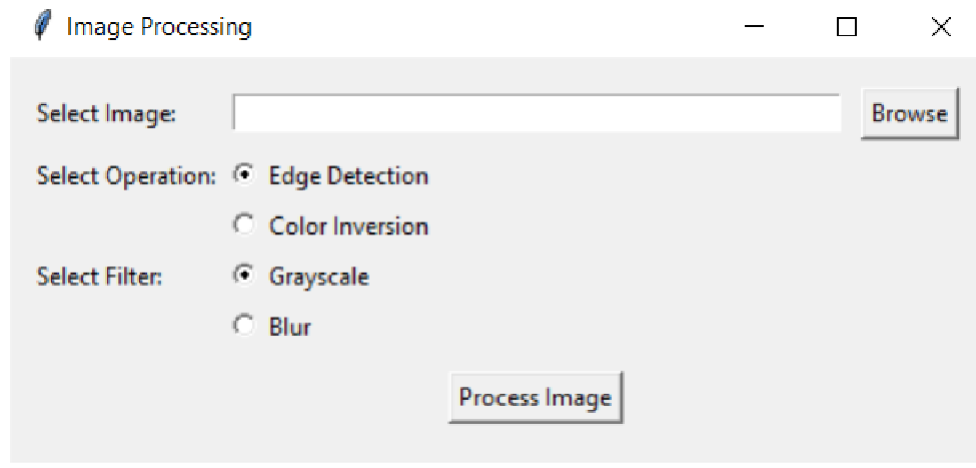


Figure 1: The Gui



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

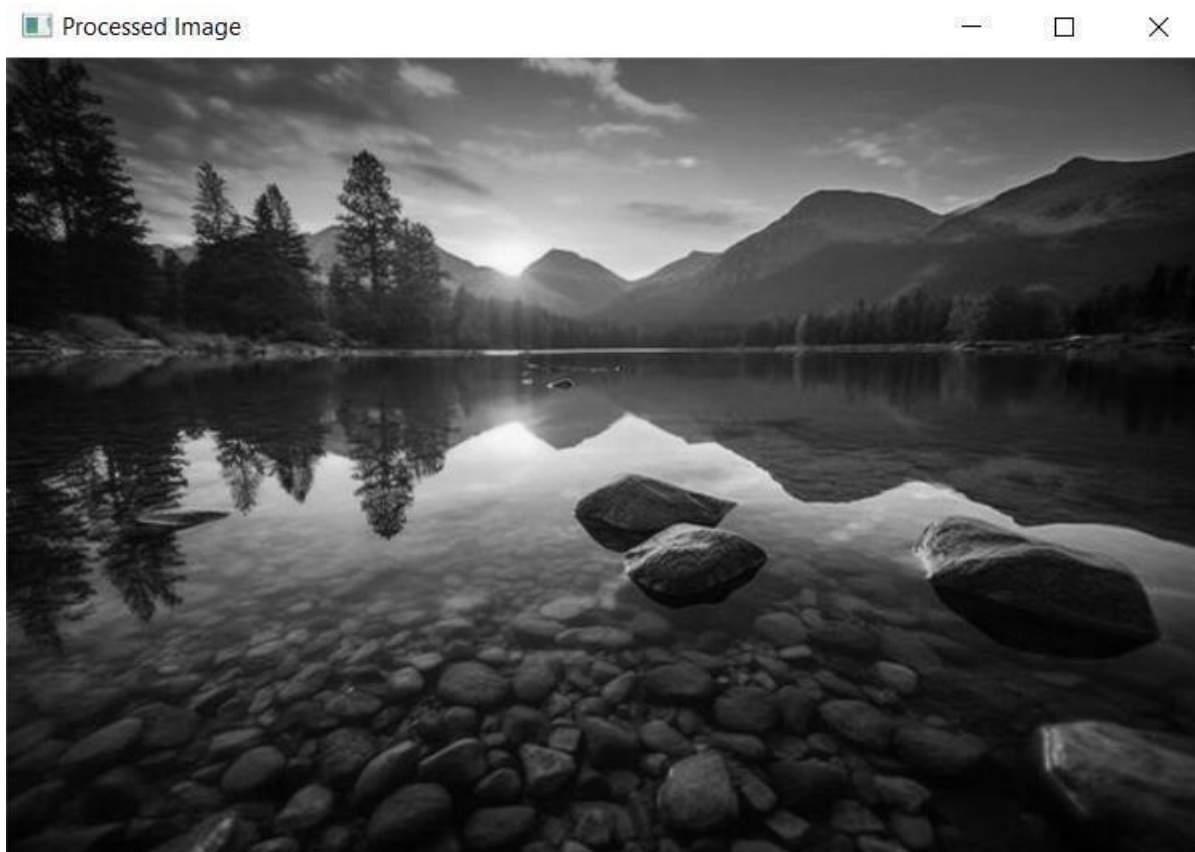


Figure 3: Processed image after processing

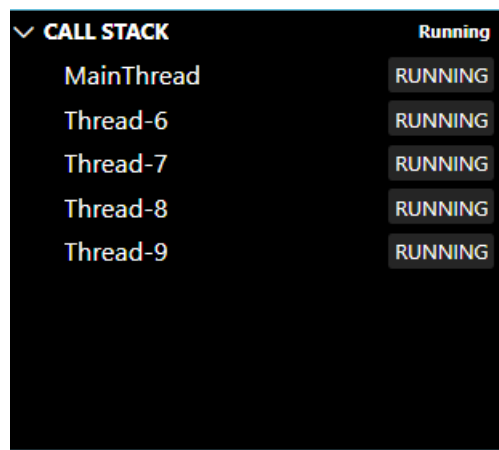


Figure 4: Many threads have been created

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

Phase 3

Distributed

The advanced processing code

```
from mpi4py import MPI
import cv2
import threading
import queue
import numpy as np

class WorkerThread(threading.Thread):
    def __init__(self, task_queue, output, rank=0):
        super().__init__()
        self.task_queue = task_queue
        self.output = output
        self.rank = rank

    def run(self):
        while True:
            task = self.task_queue.get()
            if task is None:
                print(f"thread {self.name} ready.")
                break
            image, operation = task
            result = self.process_image(image, operation)
            print(f"thread {self.name} processing {image} with {operation}.")
            self.output.put(result)

    def process_image(self, image, operation):
        img = cv2.imread(image, cv2.IMREAD_COLOR)
        if operation == 'edge_detection':
            result = cv2.Canny(img, 100, 200)
        elif operation == 'color_inversion':
            result = cv2.bitwise_not(img)
        elif operation == 'custom_processing':
            result = self.custom_processing(img)
        return result

    def custom_processing(self, img):
        hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
        h, s, v = cv2.split(hsv)
        v += 50
        final_hsv = cv2.merge((h, s, v))
        result = cv2.cvtColor(final_hsv, cv2.COLOR_HSV2BGR)
        return result

def main():
```

```

rank = MPI.COMM_WORLD.Get_rank()
size = MPI.COMM_WORLD.Get_size()
task_queue = [("img1.jpg", "edge_detection"), ("img1.jpg", "color_inversion"),
              ("img1.jpg", "custom_processing"), ("img2.jpg", "edge_detection"),
              ("img2.jpg", "color_inversion"),
              ("img2.jpg", "custom_processing")]

if size == 1:
    local_task_queue = queue.Queue()
    local_output = queue.Queue()
    for task in task_queue:
        local_task_queue.put(task)

    local_num_threads = 2

    for _ in range(local_num_threads):
        local_task_queue.put(None)

    threads = []
    for _ in range(local_num_threads):
        thread = WorkerThread(local_task_queue, local_output)
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()

    i = 0
    while not local_output.empty():
        result = local_output.get()
        filename = f"modified{i}.jpg"
        cv2.imwrite(filename, result)
        print(f"Saved {filename}")
        i += 1

else:
    if rank == 0:
        task_queue_size = len(task_queue)
        tasks_per_process = task_queue_size // (size - 1)
        remainder = task_queue_size % (size - 1)

        start_index = 0
        for dest in range(1, size):
            if dest <= remainder:
                num_of_task_for_current_process = tasks_per_process + 1;
            else:
                num_of_task_for_current_process = tasks_per_process;

```



```

        end_index = start_index + num_of_task_for_current_process;
        subtasks = task_queue[start_index:end_index]
        print(f"Sending tasks {subtasks} to process {dest}")
        MPI.COMM_WORLD.send(subtasks, dest=dest)
        start_index = end_index

results = []
for _ in range(size - 1):
    received_data = MPI.COMM_WORLD.recv(source=MPI.ANY_SOURCE)
    if isinstance(received_data, list):
        results.extend(received_data)
    else:
        results.append(received_data)

for i, result in enumerate(results):
    filename = f"modified{i}.jpg"
    cv2.imwrite(filename, result)
    print(f"Saved {filename}")

else:
    print(f"Process {rank} in the blocked worker ")
    task_queue = queue.Queue()
    output = queue.Queue()
    tasks = MPI.COMM_WORLD.recv(source=0)
    print(f"Process {rank} received task: {tasks}")
    for task in tasks:
        task_queue.put(task)

    num_threads = 1
    for _ in range(num_threads):
        task_queue.put(None)

    threads = []
    for _ in range(num_threads):
        thread = WorkerThread(task_queue, output, rank)
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()

    results_to_send = []
    while not output.empty():
        result = output.get()
        results_to_send.append(result)

```

```
MPI.COMM_WORLD.send(results_to_send, dest=0)

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

Screenshot of code output

```
PS C:\Users\dell\Desktop\Phase 3 Final\New folder (2)> & 'c:\Users\dell\Desktop\Phase 3 Final\New folder (2)\2.py'
thread Thread-7 processing img1.jpg with color_inversion.
thread Thread-6 processing img1.jpg with edge_detection.
thread Thread-7 processing img1.jpg with custom_processing.
thread Thread-7 processing img2.jpg with color_inversion.
thread Thread-6 processing img2.jpg with edge_detection.
thread Thread-6 ready.
thread Thread-7 processing img2.jpg with custom_processing.
thread Thread-7 ready.
Saved modified0.jpg
Saved modified1.jpg
Saved modified2.jpg
Saved modified3.jpg
Saved modified4.jpg
Saved modified5.jpg
```

1. Imports: The code imports necessary libraries such as MPI for message passing, cv2 for image processing using OpenCV, threading for multithreading, and queue for task queues.

2. WorkerThread Class:

- This class represents a worker thread responsible for processing images.
- The `run()` method continuously retrieves tasks from the task queue until it receives a termination signal (a `None` task).
- Each task consists of an image file name and an operation to perform on the image (e.g., edge detection, color inversion, custom processing).
- The `process_image()` method processes the image based on the specified operation.
- For custom processing, the `custom_processing()` method applies a specific transformation to the image.

3. Main Function:

- Initializes MPI environment, retrieves rank and size.
- Defines a list of tasks to be processed.
- If there's only one process (`size == 1`):
 - Tasks are processed sequentially using multiple threads.
 - Each task is put into a local task queue, and each thread processes tasks from this queue.
 - Results are written to files sequentially.
- If there are multiple processes (`size > 1`):
 - The master process (rank 0) distributes tasks to

other processes.

- Tasks are evenly distributed among processes, considering any remainder tasks.
- Each worker process receives its portion of tasks, processes them using threads, and sends back the results to the master process.
- The master process collects results from all worker processes and writes them to files.

4.Parallel Execution:

- In a single process environment, tasks are processed concurrently using multiple threads.
- In a multi-process environment, tasks are distributed among different processes, and each process uses multiple threads to process its tasks independently.

5.Output:

- For each processed image, the code saves the modified image with a filename indicating the processing order (modified0.jpg, modified1.jpg, etc.).
- Progress and completion messages are printed during execution.

Screenshots of images



Figure 1: image before edit

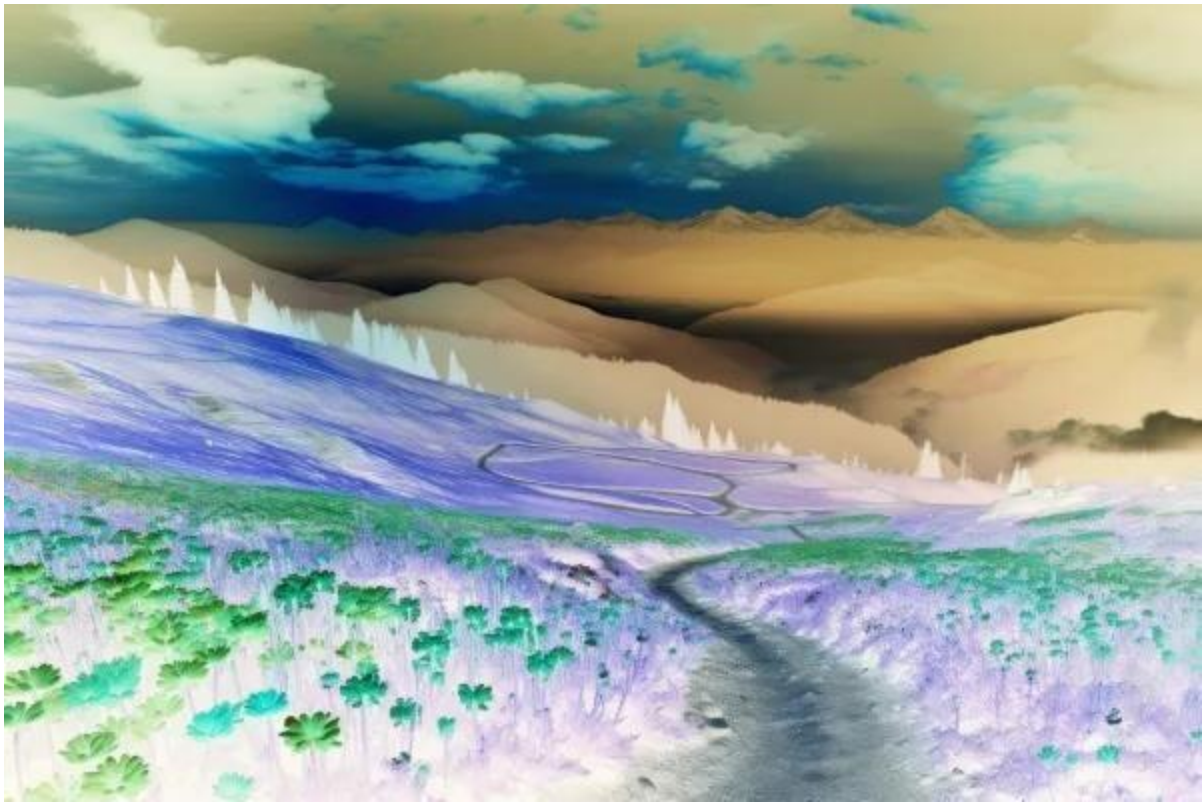


Figure 2:after edit



Figure 3:after edit also

[https://github.com/ZeyadCESS/phase-3-distributed-/tree/main/New%20folder%20\(2\)](https://github.com/ZeyadCESS/phase-3-distributed-/tree/main/New%20folder%20(2))

Final phase

Phase 4: Distributed computing

We have modified some features in the app.py file and here is the code and description of what it does

```
from mpi4py import MPI
import cv2
import threading
import queue
import numpy as np

class WorkerThread(threading.Thread):
    def __init__(self, task_queue, output, rank=0):
        super().__init__()
        self.task_queue = task_queue
        self.output = output
        self.rank = rank

    def run(self):
        while True:
            task = self.task_queue.get()
            if task is None:
                print(f"thread {self.name} ready.")
                break
            image, operation = task
            result = self.process_image(image, operation)
            print(f"thread {self.name} processing {image} with {operation}.")
            self.output.put(result)

    def process_image(self, image, operation):
        img = cv2.imread(image, cv2.IMREAD_COLOR)
        if operation == 'edge_detection':
            result = cv2.Canny(img, 100, 200)
        elif operation == 'color_inversion':
            result = cv2.bitwise_not(img)
        elif operation == 'custom_processing':
            result = self.custom_processing(img)
        return result

    def custom_processing(self, img):
        hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
        h, s, v = cv2.split(hsv)
        v += 50
        final_hsv = cv2.merge((h, s, v))
        result = cv2.cvtColor(final_hsv, cv2.COLOR_HSV2BGR)
        return result
```

```

def main():

    rank = MPI.COMM_WORLD.Get_rank()
    size = MPI.COMM_WORLD.Get_size()
    task_queue = [("img1.jpg", "edge_detection"), ("img1.jpg", "color_inversion"),
                  ("img1.jpg", "custom_processing"), ("img2.jpg", "edge_detection"),
                  ("img2.jpg", "color_inversion"),
                  ("img2.jpg", "custom_processing")]

    if size == 1:
        local_task_queue = queue.Queue()
        local_output = queue.Queue()
        for task in task_queue:
            local_task_queue.put(task)

        local_num_threads = 2

        for _ in range(local_num_threads):
            local_task_queue.put(None)

        threads = []
        for _ in range(local_num_threads):
            thread = WorkerThread(local_task_queue, local_output)
            threads.append(thread)
            thread.start()

        for thread in threads:
            thread.join()

        i = 0
        while not local_output.empty():
            result = local_output.get()
            filename = f"modified{i}.jpg"
            cv2.imwrite(filename, result)
            print(f"Saved {filename}")
            i += 1

    else:
        if rank == 0:
            task_queue_size = len(task_queue)
            tasks_per_process = task_queue_size // (size - 1)
            remainder = task_queue_size % (size - 1)

            start_index = 0
            for dest in range(1, size):
                if dest <= remainder:
                    num_of_task_for_current_process = tasks_per_process + 1;

```

```

        else:
            num_of_task_for_current_process = tasks_per_process;

            end_index = start_index + num_of_task_for_current_process;
            subtasks = task_queue[start_index:end_index]
            print(f"Sending tasks {subtasks} to process {dest}")
            MPI.COMM_WORLD.send(subtasks, dest=dest)
            start_index = end_index

    results = []
    for _ in range(size - 1):
        received_data = MPI.COMM_WORLD.recv(source=MPI.ANY_SOURCE)
        if isinstance(received_data, list):
            results.extend(received_data)
        else:
            results.append(received_data)

    for i, result in enumerate(results):
        filename = f"modified{i}.jpg"
        cv2.imwrite(filename, result)
        print(f"Saved {filename}")

else:
    print(f"Process {rank} in the blocked worker ")
    task_queue = queue.Queue()
    output = queue.Queue()
    tasks = MPI.COMM_WORLD.recv(source=0)
    print(f"Process {rank} received task: {tasks}")
    for task in tasks:
        task_queue.put(task)

    num_threads = 1
    for _ in range(num_threads):
        task_queue.put(None)

    threads = []
    for _ in range(num_threads):
        thread = WorkerThread(task_queue, output, rank)
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()

    results_to_send = []
    while not output.empty():

```

```
        result = output.get()
        results_to_send.append(result)

    MPI.COMM_WORLD.send(results_to_send, dest=0)

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

Description of what the code do:

1. Imports:

- **from mpi4py import MPI:** Imports MPI functionality for parallel processing.
- **import cv2:** Imports OpenCV library for image processing.
- **import threading:** Imports the threading module for concurrent execution of tasks.
- **import queue:** Imports the queue module for managing task queues.

2. WorkerThread Class:

- This class defines a worker thread that processes image tasks.
- Constructor (**__init__**): Initializes the thread with a task queue (**task_queue**), an output queue (**output**), and the thread's rank (**rank**).

- **run()**: Overrides the **run()** method of the **Thread** class. It continuously gets tasks from the task queue, processes them, and puts the results into the output queue until it receives a **None** task indicating the end of processing.
- **process_image()**: Processes the image based on the specified operation. It supports three operations: edge detection, color inversion, and custom processing.
- **custom_processing()**: Performs custom image processing by converting the image to HSV color space, adjusting the brightness, and converting it back to the BGR color space.

3. Main Function (main()):

- Initializes MPI and gets the rank and size of the MPI communicator.
- Defines a list of image processing tasks (**task_queue**).
- If there is only one process (**size == 1**), it executes the image processing tasks sequentially using multiple threads.
 - It divides the tasks among the threads and assigns each thread a portion of the task queue.
 - Each thread processes its assigned tasks and writes the results to image files.
- If there are multiple processes (**size > 1**):
 - Process 0 distributes the tasks among the processes and collects the results.
 - Other processes receive their tasks from process 0,

execute them using worker threads, and send the results back to process 0.

- Process 0 saves the received results to image files.

4. Execution:

- The code checks if it's running as the main script (**if `__name__ == "__main__":`**) and calls the **main()** function to start the image processing.

In summary, this code demonstrates parallel image processing using MPI. It distributes image processing tasks among multiple processes and utilizes multithreading within each process to efficiently process the tasks concurrently.

*Screenshots of the output on ubuntu

```
LINUXVMIMAGES.COM
+-----+
User Name: ubuntu
Password: ubuntu (sudo su -)
+-----+
LINUXVMIMAGES.COM
+-----+
User Name: ubuntu
Password: ubuntu (sudo su -)
Master Rank 0 IP: 192.168.229.131
Rank 1 IP: 192.168.229.131
Rank 2 IP: 192.168.229.128
Rank 3 IP: 192.168.229.128
Rank 4 IP: 192.168.229.133
ubuntu@ubuntu-2204:~/Desktop/sharedfolder$ mpirun --hostfile rank.txt -np 5 python3 constCS.py
+-----+
LINUXVMIMAGES.COM
+-----+
User Name: ubuntu
Password: ubuntu (sudo su -)
+-----+
LINUXVMIMAGES.COM
+-----+
User Name: ubuntu
Password: ubuntu (sudo su -)
Master Rank 0 IP: 192.168.229.131
ubuntu@ubuntu-2204:~/Desktop/sharedfolder$ mpirun --hostfile rank.txt -np 5 python3 constCS.py
+-----+
LINUXVMIMAGES.COM
+-----+
User Name: ubuntu
Password: ubuntu (sudo su -)
+-----+
LINUXVMIMAGES.COM
+-----+
User Name: ubuntu
Password: ubuntu (sudo su -)
Master Rank 0 IP: 192.168.229.131
Rank 1 IP: 192.168.229.131
Rank 2 IP: 192.168.229.128
Rank 3 IP: 192.168.229.128
Rank 4 IP: 192.168.229.133
ubuntu@ubuntu-2204:~/Desktop/sharedfolder$ mpirun --hostfile rank.txt -np 5 python3 constCS.py
```

```

Running ifconfig command on Rank 3
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.229.131 netmask 255.255.255.0 broadcast 192.168.229.255
    inet6 fe80::1f32:2634:abfb:cf29 prefixlen 64 scopeid 0x20<link>
    ether 00:0c:29:13:08:7a txqueuelen 1000 (Ethernet)
    RX packets 128983 bytes 166581706 (166.5 MB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 67256 bytes 6829690 (6.8 MB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 5163 bytes 535340 (535.3 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 5163 bytes 535340 (535.3 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

Running ifconfig command on Rank 2
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.229.128 netmask 255.255.255.0 broadcast 192.168.229.255
    inet6 fe80::685b:1f83:84ff:ec5e prefixlen 64 scopeid 0x20<link>
    ether 00:0c:29:b6:ab:e2 txqueuelen 1000 (Ethernet)
    RX packets 7478 bytes 4427361 (4.4 MB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 6570 bytes 886924 (886.9 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 1013 bytes 118359 (118.3 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 1013 bytes 118359 (118.3 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

```



```

ubuntu@ubuntu-2204:~/Desktop/sharedfolder$ mpirun --hostfile rank.txt -np 5 python3 constCS.py
+++++
LINUXVMIMAGES.COM
+++++
User Name: ubuntu
Password: ubuntu (sudo su -)
+++++
LINUXVMIMAGES.COM
+++++
User Name: ubuntu
Password: ubuntu (sudo su -)
Running ifconfig command on Rank 0:
Running ifconfig command on Rank 1
Running ifconfig command on Rank 4
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.229.131 netmask 255.255.255.0 broadcast 192.168.229.255
    inet6 fe80::1f32:2634:abfb:cf29 prefixlen 64 scopeid 0x20<link>
    ether 00:0c:29:13:08:7a txqueuelen 1000 (Ethernet)
    RX packets 128982 bytes 166581518 (166.5 MB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 67256 bytes 6829690 (6.8 MB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 5163 bytes 535340 (535.3 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 5163 bytes 535340 (535.3 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

Running ifconfig command on Rank 0
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.229.131 netmask 255.255.255.0 broadcast 192.168.229.255
    inet6 fe80::1f32:2634:abfb:cf29 prefixlen 64 scopeid 0x20<link>
    ether 00:0c:29:13:08:7a txqueuelen 1000 (Ethernet)
    RX packets 128982 bytes 166581518 (166.5 MB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 67256 bytes 6829690 (6.8 MB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>

```

```

ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.229.128 netmask 255.255.255.0 broadcast 192.168.229.255
    inet6 fe80::685b:1f83:84ff:ec5e prefixlen 64 scopeid 0x20<link>
    ether 00:0c:29:b6:ab:e2 txqueuelen 1000 (Ethernet)
    RX packets 7478 bytes 4427361 (4.4 MB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 6570 bytes 886924 (886.9 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 1013 bytes 118359 (118.3 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 1013 bytes 118359 (118.3 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.229.133 netmask 255.255.255.0 broadcast 192.168.229.255
    inet6 fe80::9470:9e8b:9133:b5b7 prefixlen 64 scopeid 0x20<link>
    ether 00:0c:29:23:78:1f txqueuelen 1000 (Ethernet)
    RX packets 7745 bytes 4725772 (4.7 MB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 6124 bytes 846036 (846.0 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

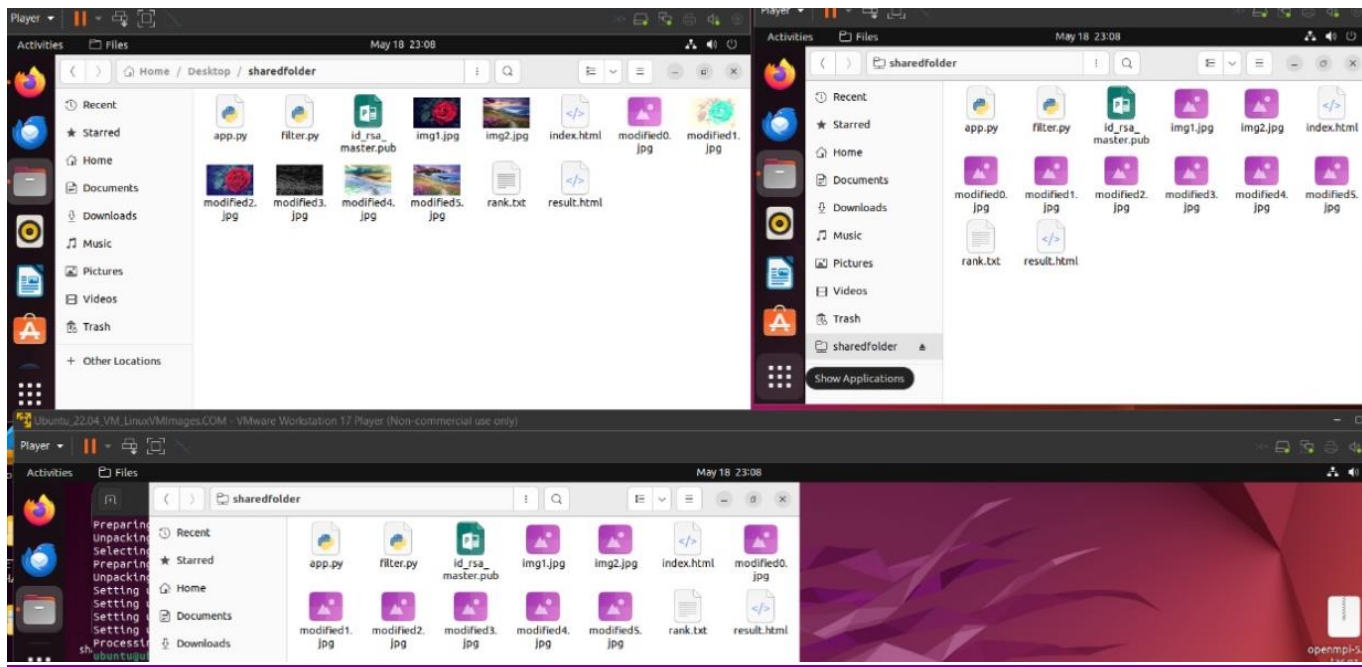
lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 1574 bytes 144668 (144.6 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 1574 bytes 144668 (144.6 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

```

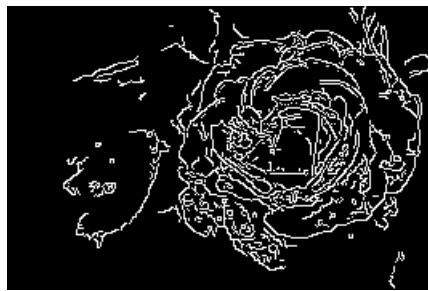
```

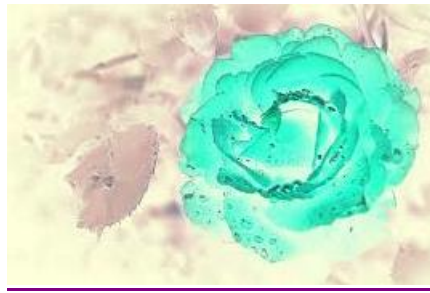
ubuntu@ubuntu-2204:~/Desktop/sharedfolder$ mpirun --hostfile rank.txt -np 3 python3 app.py
+++++
LINUXVMIMAGES.COM
+++++
User Name: ubuntu
Password: ubuntu (sudo su -)
+++++
LINUXVMIMAGES.COM
+++++
User Name: ubuntu
Password: ubuntu (sudo su -)
Process 2 in the blocked worker
Sending tasks [('img1.jpg', 'edge_detection'), ('img1.jpg', 'color_inversion'), ('img1.jpg', 'custom_processing')]
Process 1 in the blocked worker
Sending tasks [('img2.jpg', 'edge_detection'), ('img2.jpg', 'color_inversion'), ('img2.jpg', 'custom_processing')]
Process 1 received task: [('img1.jpg', 'edge_detection'), ('img1.jpg', 'color_inversion'), ('img1.jpg', 'custom_processing')]
Process 2 received task: [('img2.jpg', 'edge_detection'), ('img2.jpg', 'color_inversion'), ('img2.jpg', 'custom_processing')]
thread Thread-1 processing img1.jpg with edge_detection.
thread Thread-1 processing img1.jpg with color_inversion.
thread Thread-1 processing img1.jpg with custom_processing.
thread Thread-1 ready.
thread Thread-1 processing img2.jpg with edge_detection.
thread Thread-1 processing img2.jpg with color_inversion.
thread Thread-1 processing img2.jpg with custom_processing.
thread Thread-1 ready.
Saved modified0.jpg
Saved modified1.jpg
Saved modified2.jpg
Saved modified3.jpg
Saved modified4.jpg
Saved modified5.jpg

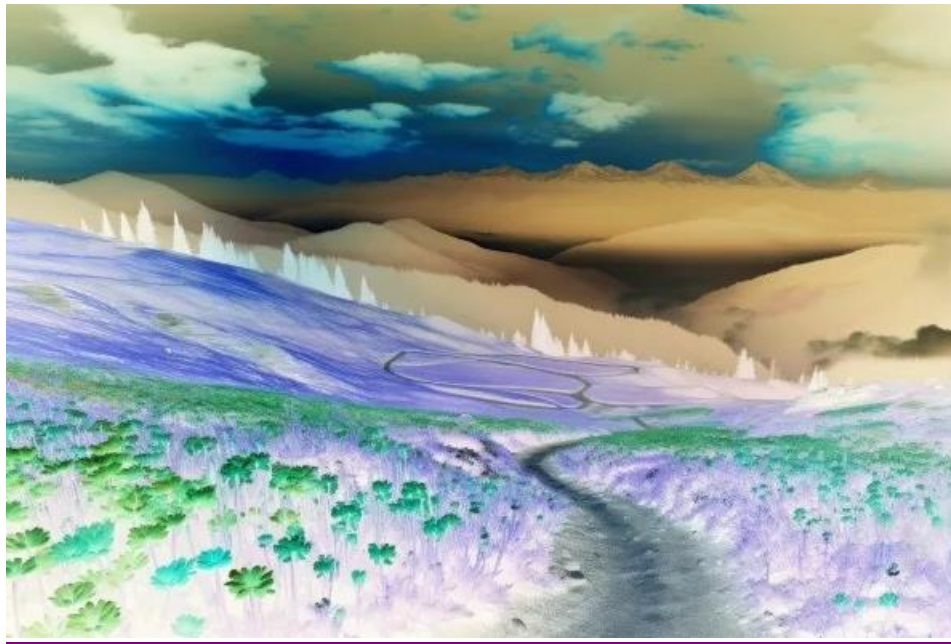
```



*Screenshots of the images







**we have implemented two VMs and here is there codes

The code of first VM

```
import pickle
import select
import socket
import threading
import image_processing_helper as img_helper
import pyopengl 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

```

The screenshot shows a code editor with the following Python code for the `VMHandler` class:

```

15     self.port = port
16     self.socket = socket
17     self.socket.setblocking(0)
18     self.gpu_context = alt_lib.create_context()
19     self.gpu_queue = alt_lib.create_queue(self.gpu_context)
20     self.kernels = {
21         'brighten': """
22         kernel void brighten( global float* v) {

```

Below the code editor is a terminal window with the following content:

```

PROBLEMS (8) OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\dell\Desktop\Distributed> & 'c:\Users\dell\AppData\Local\Programs\Python\Python311\python.exe' 'c:\Users\dell\AppData\Local\Programs\Python\Python311\Scripts\debugpy-2024.6.0-win32-x64\bundled\libs\debugpy\adapter\../..debugpy\1.py'
First virtual machine Start listening on IP:0.0.0.0, PORT:5030

```

**Description of the code:

This code implements a basic server-client architecture for image processing using Python's socket programming. Let's break down its functionality:

1. **Imports and Constants:** It imports necessary modules such as **pickle**, **select**, **socket**, **threading**, and external modules like **image_processing_helper** and **pyopencv** for image

processing and GPU computation. Constants like **IP_ADDRESS** and **PORT** are defined for server setup.

2. **VMHandler Class:** This class is responsible for handling client requests in separate threads.

- **Initialization:** It initializes with IP, port, and a socket object. It also sets the socket to non-blocking mode and creates a GPU context and queue for OpenCL operations.
- **run() Method:** This method runs in a loop to continuously handle client requests. It receives data from the client, processes it based on the specified operation (e.g., brighten, grayscale), and sends back the processed image data.
- **receive_data() Method:** This method receives data from the client. It uses **select** to check if there's data to be read from the socket within a timeout period. It then deserializes the received data using **pickle** and returns it.
- **send_data() Method:** This method sends data back to the client. It serializes the data using **pickle** and sends it over the socket.
- **process_image() Method:** This method processes the image based on the operation requested by the client. It utilizes OpenCL for GPU acceleration. Different operations like brightening, grayscale conversion, and thresholding are

implemented as OpenCL kernels.

3. **Server Setup:** It creates a TCP socket, binds it to the specified IP and port, and starts listening for incoming connections. When a client connects, it accepts the connection, creates a new **VMHandler** thread to handle the client, and starts the thread.

Overall, this code sets up a server capable of receiving image processing requests from clients, executing those requests using OpenCL for GPU acceleration, and sending back the processed results

**with every code we have provided a helper file with some imports

The code of the helper:

```
import pyopencl as cl
import numpy as np

def convert_to_grayscale(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

```

**Description of the code

This code is using the PyOpenCL library to perform various image processing operations using OpenCL kernels. Let's break down each function:

1. **convert_to_greyscale**: This function takes an OpenCL context (**ctx**), a command queue (**queue**), a single-channel image (**channel**), its height and width, and a greyscale conversion kernel (**greyscale_kernel**). It reshapes the input channel to a flat array, creates an empty array with the same shape, creates OpenCL buffers for the input and output data, builds the OpenCL program from the provided kernel, executes the kernel on the input buffer to convert the image to greyscale, copies the result back to a NumPy array, and reshapes it back to the original image dimensions before returning it.
2. **apply_intensity_kernel**: This function applies brightness or darkness adjustment to a single-channel image using OpenCL kernels. It takes similar parameters as **convert_to_greyscale**, along with a **value** parameter indicating whether to apply brightness (**value == 1**) or darkness (**value != 1**). It performs similar steps to create buffers, build the appropriate OpenCL program based on the **value**, executes the corresponding kernel, and copies the result back to a NumPy array before reshaping it to the original image dimensions.
3. **threshold_helper**: This function applies thresholding operation to an image using OpenCL. It takes an OpenCL context (**ctx**), a command queue (**queue**), the image data (**data**), a

thresholding kernel (**threshold_kernel**), and the original height and width of the image. It creates an OpenCL buffer for the image data, builds the OpenCL program from the provided kernel, executes the kernel on the buffer, copies the result back to the **data** array, reshapes it to the original image dimensions, converts it to unsigned 8-bit integers (assuming the thresholding operation outputs binary values), and returns the thresholded image.

**second VM and helper codes are similar to these two codes with little changes and here are the two codes

**second VM code:

```
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: {}".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

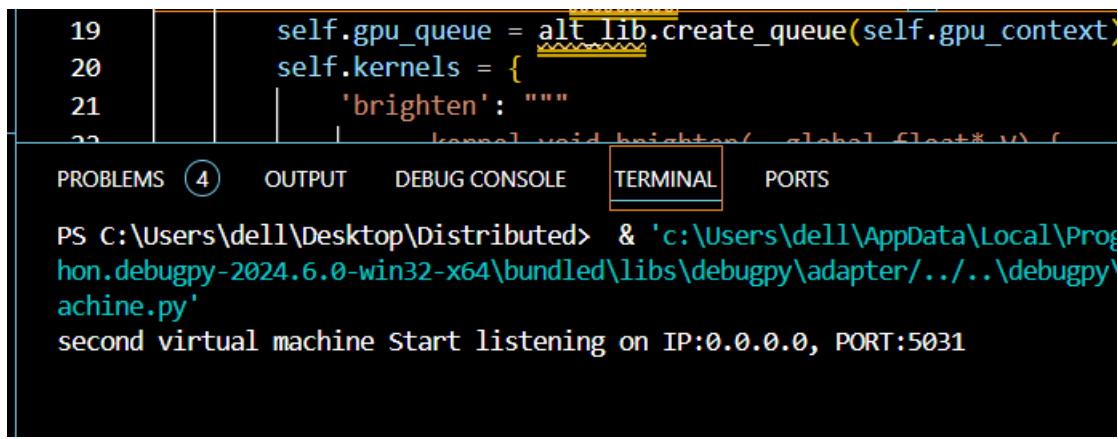
tcp_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
tcp_socket.bind((IP_ADDRESS, PORT))
print(f"second virtual machine Start listening on IP:{IP_ADDRESS}, PORT:{PORT}")
tcp_socket.listen(1)

```

```

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

```



The screenshot shows a code editor with the following Python code:

```

19 self.gpu_queue = alt_lib.create_queue(self.gpu_context)
20 self.kernels = {
21     'brighten': ""
22     kernel void brighten( __global float* v) {

```

Below the code editor is a terminal window with the following tabs: PROBLEMS (4), OUTPUT, DEBUG CONSOLE, TERMINAL, and PORTS. The terminal shows the command prompt output:

```

PS C:\Users\dell\Desktop\Distributed> & 'c:\Users\dell\AppData\Local\Programs\Python\Python310\python.exe' -c 'import sys; sys.path.append('c:\Users\dell\AppData\Local\Programs\Python\Python310\python.exe'); import vmhandler; vmhandler.VMHandler('0.0.0.0', 5031).start()'
second virtual machine Start listening on IP:0.0.0.0, PORT:5031

```

**second helper code :

```

import pyopenc1 as c1
import numpy as np

```



```

def convert_to_grayscale(ctx, queue, channel, height, width, grayscale_kernel):
    """
    Convert a single channel to grayscale 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, grayscale_kernel).build()
    program.grayscale(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)

c1.enqueue_copy(queue, data, buffer_data)
thresholded_image = data.reshape((original_height, original_width))
thresholded_image = thresholded_image.astype(np.uint8)

return thresholded_image

```

some screenshots from aws

```

(venv) ubuntu@ip-172-31-23-225:~/eslam$ python app.py
* Serving Flask app 'app'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit

```

Instances (2) [Info](#)

Find Instance by attribute or tag (case-sensitive)

Instance state = running X

Clear filters

Refresh

Connect






Instance state ▼

Actions ▼

Launch instances ▼

All states ▼

< 1 > ⚙

<input type="checkbox"/>	Name  ▼	Instance ID	Instance state ▼	Instance type ▼	Status check	Alarm status	Availability Zone ▼	Public IPv4 DNS
<input type="checkbox"/>	processing	i-062196d81f0ceb120	Running  	t2.micro	2/2 checks passed View alarms +		us-east-1c	ec2-54-196-205
<input type="checkbox"/>	project	i-0feee025be06aa7e4	Running  	t2.micro	2/2 checks passed View alarms +		us-east-1c	ec2-54-87-30-3

Select an instance

=

⚙

X

Instance summary for i-0feee025be06aa7e4 (project) [Info](#)

Updated less than a minute ago



Connect

Instance state ▼

Actions ▼

Instance ID

i-0feee025be06aa7e4 (project)

IPv6 address

–

Hostname type

IP name: ip-172-31-23-225.ec2.internal

Answer private resource DNS name

IPv4 (A)

Auto-assigned IP address

54.87.30.37 [Public IP]

IAM Role

–

IMDSv2

Required

Public IPv4 address

54.87.30.37 [open address](#)

Instance state

Running

Private IP DNS name (IPv4 only)

ip-172-31-23-225.ec2.internal

Instance type

t2.micro

VPC ID

vpc-0012b36befdba7667

Subnet ID

subnet-07ce4cc81dc868128

Private IPv4 addresses

172.31.23.225

Public IPv4 DNS

ec2-54-87-30-37.compute-1.amazonaws.com | [open address](#)

Elastic IP addresses

–

AWS Compute Optimizer finding

[Opt-in to AWS Compute Optimizer for recommendations.](#)

| [Learn more](#)

Auto Scaling Group name

–

* Management: <https://landscape.canonical.com>
* Support: <https://ubuntu.com/pro>

System information as of Sun May 19 19:45:15 UTC 2024

System load:	0.08	Processes:	105
Usage of /:	28.8% of 6.71GB	Users logged in:	0
Memory usage:	24%	IPv4 address for enX0:	172.31.23.225
Swap usage:	0%		

* Ubuntu Pro delivers the most comprehensive open source security and compliance features.

<https://ubuntu.com/aws/pro>

Expanded Security Maintenance for Applications is not enabled.

4 updates can be applied immediately.

To see these additional updates run: `apt list --upgradable`

Enable ESM Apps to receive additional future security updates.

See <https://ubuntu.com/esm> or run: `sudo pro status`

*** System restart required ***

Last login: Sun May 19 19:35:36 2024 from 18.206.107.28

ubuntu@ip-172-31-23-225:~\$

Upload Images for Processing

No file chosen

Select Operation: ▾

Processed Images

Original Images



Processed Images



Processed Images

Original Images



Processed Images



Processed Images

Original Images



Processed Images



ubuntu video:

download the file to play the video

<https://drive.google.com/drive/folders/1XdEZ3x-b1KCDJJzrzCuNoEttXF9AXeue>

download the file to play the video

another video for aws:

<https://drive.google.com/drive/folders/1-SEifSexYWGpFbmroC8pdofrhDQcp98P>

Github link:

<https://github.com/ZeyadCESS/Phase-4-distributed-final>