COMSATS UNIVERSITY ISLAMABAD,

ABBOTTABAD

Submitted By: SAIFULLAH, HANZLA

Roll Number: SP22-BCS-200

Section: 8-B

SMESTER PROJECT

Contents

1 CUDA-Based Image Object Detection System

1.1 Introduction

This project is developed as a semester project for the course *Parallel and Distributed Computing*.
The main purpose of this project is to demonstrate how parallel computing concepts can be applied to image processing using CUDA.
Instead of using machine learning, classical computer vision techniques are used to detect objects such as vehicles and buildings from an image.

The system processes a single input image and performs grayscale conversion, edge detection, and object localization using bounding boxes.

1.2 Objectives

- To understand parallel computing concepts using CUDA
- To perform image preprocessing and analysis
- To detect objects without using machine learning
- To implement an efficient and explainable image detection pipeline

1.3 Tools and Technologies Used

- Programming Language:**C++**
- Parallel Computing Platform: **CUDA**
- Image Processing Library: **OpenCV**
- Development Environment: **Google Colab (GPU Runtime)**

1.4 System Workflow

The project is divided into three main stages:

1. Grayscale Conversion
2. Edge Detection
3. Object Detection using Bounding Boxes

Each stage refines the image and prepares it for the next step.

1.5 Grayscale Conversion

In the first stage, the input color image is converted into a grayscale image.
Grayscale conversion reduces image complexity by removing color information while preserving important intensity details.
This step makes further processing faster and more efficient.

OpenCV's grayscale conversion is used to ensure stable and accurate output before applying advanced processing techniques.

1.6 Edge Detection

In the second stage, edge detection is performed on the grayscale image.
Before detecting edges, Gaussian Blur is applied to reduce noise and unwanted details.

Canny Edge Detection is then used to extract strong edges that represent object boundaries such as cars, roads, and buildings.
This step highlights important structural features in the image.

1.7 Results

The final output image shows detected vehicles and buildings marked with colored bounding boxes.
The system successfully localizes objects using classical image processing techniques without using machine learning.

Although the system is not perfect, it demonstrates reliable object detection and clearly explains parallel image processing concepts.

1.8 Advantages of the System

- No machine learning required
- Simple and explainable logic
- Demonstrates parallel computing concepts
- Easy to modify and extend
- Suitable for academic learning

1.9 Limitations

- Detection accuracy depends on lighting and image quality
- Small or overlapping objects may not always be detected

- Not suitable for real-time detection without optimization

1.10 Conclusion

This project demonstrates how CUDA and classical computer vision techniques can be used to perform object detection.
The system successfully processes images through multiple stages and detects vehicles and buildings using bounding boxes.

The project fulfills the objectives of the Parallel and Distributed Computing course and provides a strong foundation for future enhancements.

1.11 Future Enhancements

- Integration of deep learning models such as YOLO
- Real-time video processing
- Web-based user interface for user uploads

1.12 Code/Output

```
%%writefile grayscale.cu
#include <opencv2/opencv.hpp>
#include <cuda_runtime.h>
#include <iostream>
#include <chrono>

using namespace std;
using namespace cv;

/* ================= CUDA KERNEL ================= */
__global__ void grayscaleKernel(unsigned char* input,
                                unsigned char* output,
                                int width, int height)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < width * height)
    {
        int i = idx * 3;   // BGR image
        unsigned char b = input[i];
        unsigned char g = input[i + 1];
```

```
        unsigned char r = input[i + 2];

        output[idx] = (r + g + b) / 3;
    }
}

/* ================ CPU FUNCTION ================ */
void grayscaleCPU(Mat& input, Mat& output)
{
    for (int y = 0; y < input.rows; y++)
        for (int x = 0; x < input.cols; x++)
        {
            Vec3b p = input.at<Vec3b>(y, x);
            output.at<uchar>(y, x) =
                (p[0] + p[1] + p[2]) / 3;
        }
}

int main()
{
    Mat img = imread("input.jpg");
    if (img.empty())
    {
        cout << "Image not found!" << endl;
        return -1;
    }

    int width = img.cols;
    int height = img.rows;

    Mat grayCPU(height, width, CV_8UC1);
    Mat grayGPU(height, width, CV_8UC1);

    size_t imgSize = width * height * 3;
    size_t graySize = width * height;

    unsigned char *d_input, *d_output;
    unsigned char *h_output = new unsigned char[graySize];

    cudaMalloc(&d_input, imgSize);
    cudaMalloc(&d_output, graySize);

    cudaMemcpy(d_input, img.data, imgSize, cudaMemcpyHostToDevice);

    /* ---------- CPU TIME ---------- */
```

```
    auto cpuStart = chrono::high_resolution_clock::now();
    grayscaleCPU(img, grayCPU);
    auto cpuEnd = chrono::high_resolution_clock::now();

    /* ---------- GPU TIME ---------- */
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaEventRecord(start);

    int threads = 256;
    int blocks = (width * height + threads - 1) / threads;
    grayscaleKernel<<<blocks, threads>>>(d_input, d_output, width,
height);

    cudaEventRecord(stop);
    cudaEventSynchronize(stop);

    cudaMemcpy(h_output, d_output, graySize, cudaMemcpyDeviceToHost);
    memcpy(grayGPU.data, h_output, graySize);

    float gpuTime;
    cudaEventElapsedTime(&gpuTime, start, stop);

    auto cpuTime =
        chrono::duration_cast<chrono::milliseconds>(cpuEnd - cpuStart);

    cout << "CPU Time: " << cpuTime.count() << " ms" << endl;
    cout << "GPU Time: " << gpuTime << " ms" << endl;

    imwrite("gray_cpu.jpg", grayCPU);
    imwrite("gray_gpu.jpg", grayGPU);

    delete[] h_output;
    cudaFree(d_input);
    cudaFree(d_output);

    return 0;
}
```

- **!nvcc grayscale.cu -o grayscale `pkg-config --cflags --libs opencv4`**
- **!./grayscale**

After that we need to show us image for showing image we need code

```
import matplotlib.pyplot as plt
plt.imshow(img, cmap="gray")
plt.axis("off")
```

It will show us image:

%%writefile grayscale.cu

```
#include <opencv2/opencv.hpp>

#include <cuda_runtime.h>

#include <iostream>

#include <chrono>


using namespace std;

using namespace cv;


/* ================ CUDA KERNEL ================ */

__global__ void grayscaleKernel(unsigned char* input,

                                unsigned char* output,

                                int width, int height)
```

```
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < width * height)
    {
        int i = idx * 3;
        unsigned char b = input[i];
        unsigned char g = input[i + 1];
        unsigned char r = input[i + 2];
        output[idx] = (r + g + b) / 3;
    }
}

int main()
{
    /* -------- Load image safely -------- */
    Mat img = imread("input.jpg", IMREAD_COLOR);
    if (img.empty())
    {
        cout << "Image not found!" << endl;
        return -1;
    }

    if (img.type() != CV_8UC3)
        img.convertTo(img, CV_8UC3);

    int width  = img.cols;
```

```
int height = img.rows;

Mat grayGPU(height, width, CV_8UC1);

size_t imgSize  = width * height * 3;
size_t graySize = width * height;

unsigned char *d_input, *d_output;
unsigned char *h_output = new unsigned char[graySize];

memset(h_output, 255, graySize);   // proof buffer

cudaMalloc(&d_input, imgSize);
cudaMalloc(&d_output, graySize);

cudaMemcpy(d_input, img.data, imgSize, cudaMemcpyHostToDevice);

/* -------- Launch kernel -------- */
int threads = 256;
int blocks  = (width * height + threads - 1) / threads;

grayscaleKernel<<<blocks, threads>>>(d_input, d_output, width, height);

/* -------- HARD SYNC + ERROR CHECK -------- */
cudaDeviceSynchronize();
cudaError_t err = cudaGetLastError();
```

```
    if (err != cudaSuccess)
    {
        cout << "CUDA ERROR: " << cudaGetErrorString(err) << endl;
        return -1;
    }

    cudaMemcpy(h_output, d_output, graySize, cudaMemcpyDeviceToHost);
    memcpy(grayGPU.data, h_output, graySize);

    imwrite("gray_gpu.jpg", grayGPU);

    cudaFree(d_input);
    cudaFree(d_output);
    delete[] h_output;

    cout << "GPU grayscale DONE" << endl;
    return 0;
}
```

After that we need to to find out the edages of the image ,to do this we need a code and put this into new cell

```
%%writefile edges.cpp
#include <opencv2/opencv.hpp>
#include <iostream>
```

```
using namespace cv;
using namespace std;

int main()
{
   Mat gray = imread("gray_gpu.jpg", IMREAD_GRAYSCALE);
   if (gray.empty())
   {
      cout << "gray_gpu.jpg not found\n";
      return -1;
   }

   // 🔍 Noise kam karne ke liye blur
   GaussianBlur(gray, gray, Size(9,9), 2.0);

   Mat edges;
  Canny(gray, edges, 160, 320);  // strong edges only

   imwrite("edges_gpu.jpg", edges);
   cout << "Edges generated\n";

   return 0;
}
```

!g++ edges.cpp -o edges `pkg-config --cflags --libs opencv4`

!./edges

To show edges we need to write some code :

from PIL import Image

import matplotlib.pyplot as plt

plt.imshow(Image.open("edges_gpu.jpg"), cmap="gray")

plt.axis("off")

It will give us a image:

```
MIN: 0 MAX: 255
(np.float64(-0.5), np.float64(3583.5), r
```

We can see clearly edges on this image

Now we add boxes on the object to clearly to see object

%%writefile sobel.cu

#include <opencv2/opencv.hpp>

#include <cuda_runtime.h>

```
#include <iostream>

using namespace std;
using namespace cv;

/* ========== CUDA SOBEL KERNEL ========== */
__global__ void sobelKernel(unsigned char* input,
                            unsigned char* output,
                            int width, int height)
{
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;

  if (x > 0 && x < width-1 && y > 0 && y < height-1)
  {
    int idx = y * width + x;

    int gx =
      -input[(y-1)*width + (x-1)] + input[(y-1)*width + (x+1)] +
      -2*input[y*width + (x-1)]   + 2*input[y*width + (x+1)] +
      -input[(y+1)*width + (x-1)] + input[(y+1)*width + (x+1)];

    int gy =
      -input[(y-1)*width + (x-1)] -2*input[(y-1)*width + x] -input[(y-1)*width + (x+1)] +
       input[(y+1)*width + (x-1)] +2*input[(y+1)*width + x] +input[(y+1)*width + (x+1)];
```

```
        int edge = abs(gx) + abs(gy);
        if (edge > 255) edge = 255;

        output[idx] = edge;
    }
}

/* ========== MAIN FUNCTION ========== */
int main()
{
    Mat gray = imread("input.jpg", IMREAD_GRAYSCALE);
    if (gray.empty())
    {
        cout << "Image not found!" << endl;
        return -1;
    }

    int width = gray.cols;
    int height = gray.rows;

    Mat edges(height, width, CV_8UC1);

    size_t size = width * height;

    unsigned char *d_input, *d_output;
    cudaMalloc(&d_input, size);
```

```
    cudaMalloc(&d_output, size);

    cudaMemcpy(d_input, gray.data, size, cudaMemcpyHostToDevice);

    dim3 threads(16, 16);
    dim3 blocks((width + 15) / 16, (height + 15) / 16);

    sobelKernel<<<blocks, threads>>>(d_input, d_output, width, height);
    cudaDeviceSynchronize();

    cudaMemcpy(edges.data, d_output, size, cudaMemcpyDeviceToHost);

    imwrite("edges_gpu.jpg", edges);

    cudaFree(d_input);
    cudaFree(d_output);

    cout << "Edge detection completed" << endl;
    return 0;
}
```

this will give us a image:

Now we can see a clearly edges on the object.