SCHOOL OF COMPUTER SCIENCE ENGINEERING AND INFORMATION SYSTEM

Course Project -B.TECH (IT) – Winter Semester - 2024-25

BITE301L-Computer Architecture and Organization

VIT®
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

COMPARATIVE ANALYSIS OF SERIAL AND PARALLEL IMPLEMENTATIONS OF GAUSSIAN BLURRING, OTSU'S THRESHOLDING AND EDGE DETECTION ALGORITHM (CUDA AND PYTORCH)

RAMYA M 23BIT0329

ELAKIYA R 23BIT0357

COMPARATIVE ANALYSIS OF SERIAL AND PARALLEL IMPLEMENTATIONS OF GAUSSIAN BLURRING, OTSU' S THRESHOLDING AND EDGE DETECTION ALGORITHM (CUDA AND PYTORCH)

TABLE OF CONTENTS

1.ABSTRACT

We present the implementation and performance analysis of Gaussian Blur, Sobel Edge Detection, Roberts Edge Detection, Prewitt Filter, and Otsu's Thresholding—core image processing algorithms widely used in medical imaging applications, such as edge enhancement in MRI/CT scans, noise suppression in X-ray images. These algorithms rely heavily on 2D convolution operations, which are computationally intensive and often limited by the performance of single-threaded systems.To overcome these limitations, we explore the benefits of parallelization by implementing each algorithm on two parallel systems: multi-core Central Processing Units (CPUs) using PyTorch and Graphics Processing Units (GPUs) using CUDA via CuPy. We also include a serial CPU implementation as a baseline for performance comparison. All experiments are conducted on Google Colaboratory (Colab).Our study aims to provide a comprehensive overview of the speedup and execution time improvements achieved through parallel processing for different image sizes. The results demonstrate that parallel implementations, particularly on GPUs, significantly outperform serial versions, offering enhanced performance critical for real-time and high-resolution image processing tasks.

2.INTRODUCTION

The performance efficiency in digital image processing is becoming increasingly crucial, especially when handling high-resolution images or real-time systems. As image processing operations involve intensive computations over large datasets, optimizing for speed and scalability is essential. Recent developments have focused on shifting from traditional serial computation methods to parallel computing models to meet these performance demands. This paper presents a comparative analysis of three fundamental image processing algorithms—Gaussian Blurring, Otsu's Thresholding, and Edge Detection—implemented using three computational models: Serial CPU, Parallel CPU using PyTorch, and GPU acceleration using CUDA in Google Colaboratory platform.

These image processing techniques are vital in tasks such as noise reduction, feature extraction, and image segmentation, particularly in domains like medical imaging and computer vision. Each algorithm utilizes a kernel-based approach where each pixel is processed using computations applied to its local neighborhood. In the GPU implementation using CUDA, parallelization is achieved by launching thousands of threads, where each thread handles a single pixel or group of pixels. This allows for massive parallel processing, offering significant speedup for large image sizes. On the other hand, PyTorch-based CPU implementations leverage multi-threaded vector operations, achieving moderate performance improvements over serial processing, which is used here as a baseline.

The objective of this study is to evaluate and compare the performance of these implementations using key metrics such as execution time, speedup, and computational efficiency across different image dimensions. The paper demonstrates how CUDA provides the highest speedup due to its ability to handle parallel computations efficiently at the thread level, while PyTorch-based CPU parallelism offers benefits over standard serial computation but remains limited by the number of available cores

ALGORITHM 1: GAUSSIAN BLUR ALGORITHM

1. Input image
2. Read the image into a 2D array (grayscale or RGB channel-wise)
3. Define the Gaussian kernel (e.g., 3×3)
4. For each pixel in the image, apply the Gaussian kernel:

    - Multiply the kernel with the 3×3 neighborhood of the pixel
    - Sum all the values
    - Normalize by dividing by the kernel weight (16)

5. Assign the result to the corresponding pixel in the output image
6. Repeat for all pixels (excluding borders or using padding)
7. Output the blurred image

Gaussian Blur is well-suited for parallelization as each pixel's output is computed independently using its 3×3 neighborhood. In GPU (CUDA) parallelization, each pixel is handled by a separate thread within a grid-block structure. The Gaussian kernel is loaded into shared or constant memory to reduce latency, and each thread performs convolution (multiplication, summation, and normalization) on its assigned pixel. On the CPU side, libraries like PyTorch or OpenMP divide the image across threads row-wise or block-wise, using vectorized operations for efficient parallel processing. This parallel approach eliminates the need for serial looping, allowing multiple pixels to be blurred simultaneously, resulting in significant speedup, especially for high-resolution images.

ALGORITHM 2: SOBEL EDGE DETECTION ALGORITHM

1.Input image

2. Read the image into a 2D array

3.Apply Kernel 1 on each pixel to get Gx

4.Apply Kernel 2 on each pixel to get Gy

5.Computer absolute value of each pixel in Gx

6.Computer absolute value of each pixel in Gy in same format give me for gaussain blur

7.Determine threshold value using Manhattan distance

8.Convert pixels with values more than threshold to black, otherwise white

9.Output image

Usually, the threshold value is determined by Euclidean distance formula as shown by formula (1)

$$|G| = \sqrt{G_x^2 + G_y^2} \qquad (1)$$

But to achieve better performance, we will be compromising with the Manhattan distance formula as shown in formula (2).

$$|G| = |G_x| + |G_y| \qquad (2)$$

For the Sobel edge detection, the two 3x3 kernels used are represented in Figures 1 and 2.

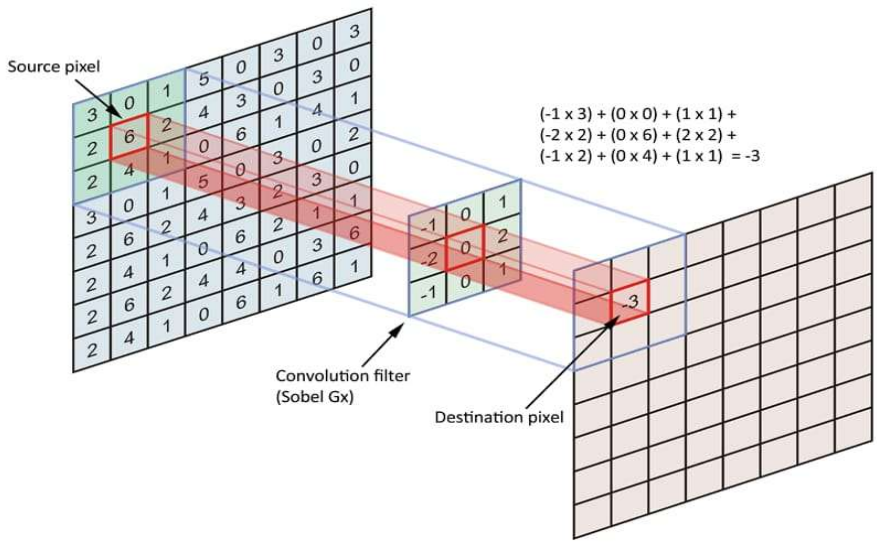
-1	0	1
-2	0	2
-1	0	1

Figure 1 - ($G_{x)}$

1	2	1
0	0	0
-1	-2	-1

Figure 2- ($G_{y)}$

2D IMAGE CONVOLUTION



ALGORITHM 3: PREWITT EDGE DETECTION ALGORITHM

1.Input image

2. Read the image into a 2D array

3.Apply Kernel 1 on each pixel to get Gx

4.Apply Kernel 2 on each pixel to get Gy

5.Computer absolute value of each pixel in Gx

6.Computer absolute value of each pixel in Gy in same format give me for gaussain blur

7.Determine threshold value using Manhattan distance

8.Convert pixels with values more than threshold to black, otherwise white

9.Output image

For the Prewitt edge detection, the two 3x3 kernels used are represented in Figures 1 and 2.

-1	0	+1
-1	0	+1
-1	0	+1

Figure 1 - ($G_{x)}$

+1	+1	+1
0	0	0
-1	-1	-1

Figure 2- ($G_{y)}$

ALGORITHM 4: ROBERTS EDGE DETECTION ALGORITHM

1.Input image

2. Read the image into a 2D array

3.Apply Kernel 1 on each pixel to get Gx

4.Apply Kernel 2 on each pixel to get Gy

5.Computer absolute value of each pixel in Gx

6.Computer absolute value of each pixel in Gy in same format give me for gaussain blur

7.Determine threshold value using Manhattan distance

8.Convert pixels with values more than threshold to black, otherwise white

9.Output image

For the Roberts edge detection, the two 3x3 kernels used are represented in Figures 1 and 2.

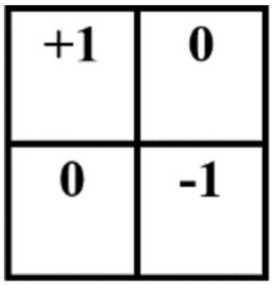


Figure 1 - (G_x)

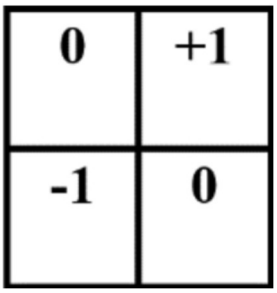


Figure 2- (G_y)

In Sobel, Prewitt, and Roberts edge detection, parallelization is achieved by processing each pixel independently using convolution with predefined kernels (Gx and Gy). On the GPU, each pixel is assigned to a separate thread that applies both kernels, computes the gradient magnitude, and applies thresholding. On the CPU, image rows or blocks are distributed among threads for concurrent execution. Since each pixel's operation is local and does not depend on others, these algorithms are ideal for parallel processing, resulting in faster execution, especially on large images.

ALGORITHM 5: OTSU THRESHOLDING ALGORITHM

1. Input grayscale image
2. Read the image into a 2D array
3. Calculate histogram of pixel intensities (0–255)
4. Compute probability of each intensity level
5. Iterate over all possible thresholds (t = 0 to 255)
6. For each threshold t:
    - Divide pixels into two classes: foreground (≥ t) and background (< t)
    - Compute class probabilities and class means
    - Compute inter-class variance
7. Select threshold t* that maximizes inter-class variance
8. Apply threshold t* to the image:
    - Pixel ≥ t* → 255 (white)
    - Pixel < t* → 0 (black)
9. Output binary segmented image

In Otsu's Thresholding, parallelization is mainly applied during histogram calculation and threshold evaluation. On the GPU, each thread can process a subset of pixels to compute the histogram in parallel using shared memory and atomic operations. Multiple threads can also evaluate possible threshold values simultaneously to find the one that maximizes between-class variance. On the CPU, threads can compute partial histograms and combine them, followed by parallel threshold search. Since these steps involve independent computations over pixel intensities, parallel execution significantly reduces processing time, especially for high-resolution images.

3.LITERATURE REVIEW:

Sno	TITLE	KEY_FINDINGS
1	Performance Analysis of GPU V/S CPU for Image Processing Applications : B. N. Manjunatha Reddy , Dr. Shanthala S., Dr. B. R. VijayaKumar	• The study compares the performance of **GPU vs. CPU for image processing and filtering** • Sequential C (CPU) and parallel CUDA (GPU) implementations of edge detection and image intensity are compared • the research results shows that the GPU implementation can achieve a **speed up** of more than **60%** of time in comparison with CPU implementation of image processing. • **GPU is found to be more suitable for large-scale, data-parallel, high-density computations** in image processing. • Parallel execution on GPUs fully utilizes processing resources to improve speedup. • Future work will consider the impact of CUDA memory transfer overhead between host and device, which may reduce speedup.
2	Performance Comparison Between OpenCV Built in CPU and GPU Functions on Image Processing Operations Authors:Hangun and Eyecioglu	• The study measures execution time for image resizing, Canny edge detection, Otsu's thresholding, and histogram equalization. • OpenCV's built-in CPU vs. GPU functions (CUDA-supported) are compared for performance analysis. • For large image sizes, the GPU achieves a speedup of approximately **10×** for Otsu's thresholding, **6.25×** for histogram equalization, **1.29×** for edge detection, and **2.27×** for image resizing compared to the CPU. • GPU benefits become more noticeable as image size increases. • Future work will focus on using CUDA Toolkit's native functions instead of OpenCV's built-in GPU functions for better performance evaluation.
3	Gaussian Blur through Parallel Computing Authors: Nahla M. Ibrahim, Ahmed Abou ElFarag and Rania Kadry	• This paper aims to give an overview on the performance enhancement of the parallel systems on image convolution using Gaussian blur algorithm. • Google Colaboratory was utilized for running the code on GPUs due to its free access.

		• GPU **processing time is nearly zero** compared to CPU processing time • Increasing CPU threads improves performance but cannot match the speed of the GPU. • Gaussian blur filtering achieves much better performance on GPUs. • As future work, Investigating and optimizing image transfer time between CPU and GPU for better speedup.
4	Real-time parallel image processing applications on multicore CPUs with OpenMP and GPGPUwith CUDA Authors: Semra Aydin, Refik Samet and Omer Faruk Bay	• A multicore CPU with OpenMP and a GPGPU with CUDA were used for thresholding military cases on eight real images. • Increasing the chunk size in OpenMP reduced execution time by **4×** compared to serial computing. • Four thresholding techniques (SISP, SIMP, MISP, and MIMP) were implemented, with **MIMP achieving the highest speedup of 71× (without transmission time) and 17× (with transmission time)** compared to serial execution. • Significant speedup reduction is observed when including transmission time due to CPU-GPU data transfer overhead. • Implementations on **GeForce, Tesla K20, and Tesla K40** showed that increasing core count improves speedup. • **Tesla K40 performed the best,** with speedups of **35× and 12× (SISP), 36× and 13× (SIMP), 54× and 16× (MISP), and 71× and 17× (MIMP)** without and with transmission time, respectively. • Another point with Tesla was that, by increasing the image resolution, the speedup rate increased • As a result, users are suggested to use Tesla K40 GPU and Multiple image transmission with multiple pixel processing to get the maximum performance.
5	Image Segmentation Using OpenMP and Its Application in Plant Species Classification	• The execution time of fine-grain and coarse-grain parallelism was analyzed for the Canny Edge Detector and Otsu's Method on a quad-core CPU for plant species classification.

	Authors: M Nordin A Rahman, Ahmad Fakhri Ab. Nasir, Nashriyah Mat and A Rasid Mamat	• Coarse-grain parallelism is more efficient, as it splits data at the start of execution, minimizing thread synchronization overhead. • Fine-grain parallelism suffers from higher overhead due to frequent creation of parallel regions. • For Canny Edge Detection ▪ Fine-grain parallelism: 2.5× speedup with 2 cores, 3.2× with 4 cores. ▪ Coarse-grain parallelism: 2.7× speedup with 2 cores, 3.5× with 4 cores. ▪ Ideal speedup: 3× with 2 cores, 4× with 4 cores. • For Otsu's Method ▪ Fine-grain parallelism: 2.4× speedup with 2 cores, 3.1× with 4 cores. ▪ Coarse-grain parallelism: 2.6× speedup with 2 cores, 3.4× with 4 cores. ▪ Ideal speedup: 3× with 2 cores, 4× with 4 cores. • Canny Edge Detection achieves slightly better speedup than Otsu's Method. • The actual speedup (fine-grain and coarse-grain) is below the ideal due to parallelization overheads.
6	A Novel Parallel Approach to Image Processing for High-Performance Computing Authors: Mohammed W. Al-Neam	• Parallel computing significantly reduces processing time, making it ideal for handling large-scale image datasets. • The proposed method achieves higher **speedup (4.0×)** and efficiency compared to existing parallel approaches. • CPU (OpenMP) and GPU (CUDA) parallelization are combined, optimizing resource utilization. • Data decomposition and load balancing techniques enhance system performance. • Processing speed improves by 37.5% compared to previous methods, showing scalability. • Future research will focus on hybrid architectures, energy efficiency, and machine learning integration for further optimizations.
7	SWOT Analysis of Parallel Processing APIs	

	- CUDA, OpenCL, OpenMP and MPI and their Usage in Various Companies Authors: Shajil Kumar P. A. 1 & Srinivasa Rao Kunte	• CUDA, OpenCL, OpenMP, and MPI are widely used parallel processing APIs, each with unique strengths and limitations. • SWOT analysis identifies Strengths, Weaknesses, Opportunities, and Threats of these APIs from an industry perspective. 1. Open MP Strength:easier to code and compile Weakness: can execute openmp prohrams in shared memory computer Threats:limited scalability Opportunities:serial region and parallel regions Support multiple type of synchronization 2.CUDA Strength: shared memory Weakness:minimum unit block of 32 threads needed Opportunities: improved performance on downloads Threats:hardware dependency to NVIDIA hardware
8	Parallel Image Processing: Taking Grayscale Conversion Using OpenMP as an Example Authors:AlHumaidan, B. , Alghofaily, S. , Qhahtani, M. , Oudah, S. and Nagy, N.	• The paper explores parallel image processing with a focus on grayscale conversion using OpenMP. By distributing tasks across multiple CPU cores, the study demonstrates significant execution time reduction while highlighting scalability challenges as the number of cores increases. • Parallel computing enhances efficiency in image processing, particularly in grayscale conversion. • OpenMP enables effective multi-core parallelization, significantly reducing computation time. • Speedup gains are more evident for larger images, proving parallelization benefits. • Increasing the number of cores doesn't always improve efficiency due to load balancing issues. • Communication overhead and thread management impact parallel performance. • Careful optimization of workload distribution is crucial for maximizing OpenMP's potential. • Future work should focus on dynamic scheduling strategies and hybrid CPU-GPU approaches for better scalability

9	Digital image processing using parallel computing based on CUDA technology Authors: I P Skirnevskiy , A V Pustovit , M O Abdrashitova	• GPU-based parallel computing significantly improves image processing speed compared to CPU execution. • CUDA technology enables efficient noise removal, essential for CT image segmentation. • Parallelization of the NLM algorithm on GPUs reduces execution time by 5.6× to 6×. • GPU-accelerated filtering achieves better performance than multi-core CPU processing. • Balancing CPU and GPU usage affects performance, with higher GPU workloads leading to faster execution. • Memory transfer between CPU and GPU introduces overhead, requiring careful optimization. • Future work aims to improve GPU memory management and explore hybrid CPU-GPU approaches for better efficiency.
10	Image Segmentation Using OpenMP and Its Application in Plant Species Classification Authors: M Nordin A Rahman, Ahmad Fakhri Ab. Nasir1, Nashriyah Mat2 and A Rasid Mamat1	• Analyzes performance with different percentages of CPU and GPU usage. • Increasing GPU usage significantly improves performance. • Two CPU threads are inefficient, performing worse than a single thread. • Partial GPU usage (½ CPU + ½ GPU) already provides notable speedup. • Higher GPU allocation further accelerates execution. • Full GPU execution achieves the highest speedup of 6.16x over a single CPU thread. • Maximum acceleration is achieved when more independent computations are offloaded to the GPU.

4.PROPOSED METHOD

In our experiment, we utilized Google Colaboratory (Colab) to implement and benchmark five image processing algorithms: Gaussian Blur, Sobel,Prewitt and Robert Edge detection and Otsu's Thresholding. Google Colaboratory is a free, cloud-based Jupyter notebook environment that allows seamless execution of Python code and GPU acceleration via NVIDIA CUDA, making it an ideal platform for our parallel computing experiments.

For GPU-based execution, we used CuPy, a NumPy-like library accelerated by CUDA. CuPy allows writing custom CUDA kernels in raw C/C++ and integrating them directly into Python code. In our case, we defined and launched CUDA kernels for Gaussian Blur, and leveraged CuPy for data management and GPU computation. We used the NVIDIA Tesla T4 GPU (available in Colab), which has 2560 CUDA cores and 40 Streaming Multiprocessors (SMs) with 16 GB of VRAM, running CUDA Toolkit version 10.1.

For the CPU-based parallel execution, we employed PyTorch and specified the number of threads explicitly using torch.set_num_threads(). PyTorch provides high-level, efficient tensor operations and convolution functions on the CPU, enabling us to simulate OpenMP-style parallelism in Python. This enabled us to compare the speedup and efficiency of multi-threaded CPU versus GPU execution. The serial version of the algorithm used scipy.signal.convolve2d() for each channel independently, with no parallel acceleration.

The Tesla T4 GPU follows the Turing architecture and provides significant acceleration for HPC workloads. We executed the kernel using (16,16) CUDA blocks and an appropriate grid size based on the image resolution. The image data was first transferred from host (CPU) to device (GPU), and the output was copied back after the operation. GPU memory was managed using CuPy's memory pool.

The CPU backend was powered by Intel Xeon processors available in Google Colab, using 2 CPU cores (2 threads total). We ensured that CPU and GPU computations were as consistent as possible in terms of kernel size, input image, and convolution method to provide a fair benchmarking comparison.

all the mentioned algorithms performs edge detection (Roberts, Sobel, Prewitt), Gaussian blur filtering, and Otsu's thresholding using three distinct computational approaches: serial CPU (using NumPy + SciPy or Numba), parallel CPU (using PyTorch), and GPU acceleration (using CuPy or custom CUDA kernels). These methods are applied to grayscale images, with the following core objectives and steps:

1. Image Upload & Preprocessing: A grayscale image is uploaded and processed for all algorithms. This includes converting the image to grayscale (if necessary) and then applying edge detection or thresholding techniques.
2. Edge Detection Methods: Various edge detection algorithms (Roberts, Sobel, Prewitt) are implemented using different computational methods:

    - Serial CPU: Using standard NumPy operations or combining with Numba for optimization.
    - Parallel CPU: Using PyTorch for efficient computation, leveraging vectorized operations.

- GPU Acceleration: Using CuPy with custom CUDA kernels to exploit GPU parallelism for high-performance processing.

3. Gaussian Blur Filtering: The script applies a Gaussian filter with varying kernel sizes to assess the impact of kernel complexity on performance across different platforms.
4. Performance Measurement: The execution time for each method is measured, and the speedup achieved by parallel and GPU implementations is calculated. The key metrics are:

    - Serial to Parallel Speedup (CPU): Measures how much faster the parallel CPU approach (PyTorch) is compared to the serial CPU approach (NumPy).
    - CPU vs serial speedup given by serial execution time/parallel execution time (CPU)
    - Serial to GPU Speedup: Measures how much faster the GPU approach (CuPy) is compared to the serial CPU approach
    - GPU vs serial speedup given by serial execution time/parallel execution time (GPU)
    - Efficiency is given by speedup / no of processing units accordingly for CPU and GPU

5.RESULTS

1.GAUSSIAN BLUR ALGORITHM

CODE:

```
import cv2
import numpy as np
import torch
import cupy as cp
import time
import matplotlib.pyplot as plt
from scipy.signal import convolve2d
from google.colab import files
uploaded = files.upload()
image_path = next(iter(uploaded))
image = cv2.imread(image_path)
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
height, width = image_rgb.shape[:2]
CPU_CORES = 2
torch.set_num_threads(CPU_CORES)
def gaussian_kernel(size=21, sigma=5.0):
    ax = np.linspace(-(size // 2), size // 2, size)
    xx, yy = np.meshgrid(ax, ax)
    kernel = np.exp(-(xx**2 + yy**2) / (2.0 * sigma**2))
    return kernel / np.sum(kernel)
def gaussian_blur_serial(image, kernel):
```

```
    blurred = np.zeros_like(image, dtype=np.float32)
    for c in range(3):
        blurred[..., c] = convolve2d(image[..., c], kernel, mode='same', boundary='symm')
    return np.clip(blurred, 0, 255).astype(np.uint8)
def gaussian_blur_pytorch_cpu(image, kernel, kernel_size):
    image_tensor = torch.tensor(image.transpose(2, 0, 1), dtype=torch.float32)
    kernel_tensor = torch.tensor(kernel, dtype=torch.float32).unsqueeze(0).unsqueeze(0)
    blurred = []
    for c in range(3):
        input_c = image_tensor[c].unsqueeze(0).unsqueeze(0)
        padded = torch.nn.functional.pad(input_c, (kernel_size//2,)*4, mode='reflect')
        out = torch.nn.functional.conv2d(padded, kernel_tensor)
        blurred.append(out.squeeze().numpy())
    blurred_img = np.stack(blurred, axis=2)
    return np.clip(blurred_img, 0, 255).astype(np.uint8)
raw_kernel_code = r'''
extern "C" __global__
void gaussian_blur(const float* img, float* out, const float* kernel,
                   int width, int height, int channels, int ksize) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    int half_k = ksize / 2;
    if (x >= width || y >= height) return;
    for (int c = 0; c < channels; ++c) {
        float val = 0.0;
        for (int ky = -half_k; ky <= half_k; ++ky) {
            for (int kx = -half_k; kx <= half_k; ++kx) {
                int ix = min(max(x + kx, 0), width - 1);
                int iy = min(max(y + ky, 0), height - 1);
                int img_idx = (iy * width + ix) * channels + c;
                int k_idx = (ky + half_k) * ksize + (kx + half_k);
                val += img[img_idx] * kernel[k_idx];
            }
        }
        int out_idx = (y * width + x) * channels + c;
        out[out_idx] = val;
    }
}
'''
module = cp.RawModule(code=raw_kernel_code)
raw_gaussian_blur = module.get_function("gaussian_blur")
def gaussian_blur_cupy_raw(image, kernel, kernel_size):
    height, width, channels = image.shape
    img_cp = cp.asarray(image, dtype=cp.float32)
    out_cp = cp.zeros_like(img_cp)
    img_flat = img_cp.ravel()
    out_flat = out_cp.ravel()
    kernel_flat = cp.asarray(kernel, dtype=cp.float32).ravel()
```

```
    block = (16, 16)
    grid = ((width + block[0] - 1) // block[0],
            (height + block[1] - 1) // block[1])
    raw_gaussian_blur(grid, block,
                      (img_flat, out_flat, kernel_flat,
                       np.int32(width), np.int32(height),
                       np.int32(channels), np.int32(kernel_size)))
    cp.cuda.Device(0).synchronize()
    cp._default_memory_pool.free_all_blocks()
    return cp.asnumpy(cp.clip(out_cp, 0, 255)).astype(np.uint8)
kernel_sizes = [7, 13, 15, 17, 31]
results = []
GPU_SMS = 40
for size in kernel_sizes:
    sigma = size / 2.0
    kernel_np = gaussian_kernel(size, sigma)

    _ = gaussian_blur_pytorch_cpu(image_rgb, kernel_np, size)
    _ = gaussian_blur_cupy_raw(image_rgb, kernel_np, size)
    start = time.time()
    blur_serial = gaussian_blur_serial(image_rgb, kernel_np)
    time_serial = time.time() - start
    start = time.time()
    blur_pytorch = gaussian_blur_pytorch_cpu(image_rgb, kernel_np, size)
    time_pytorch = time.time() - start
    start = time.time()
    blur_gpu = gaussian_blur_cupy_raw(image_rgb, kernel_np, size)
    time_gpu = time.time() - start
    speedup_pytorch = time_serial / time_pytorch
    speedup_cupy = time_serial / time_gpu
    cpu_eff = speedup_pytorch / CPU_CORES
    gpu_eff_sm = speedup_cupy / GPU_SMS
    cpu_gpu_ratio = time_pytorch / time_gpu
    results.append({
        'kernel_size': f"{size}x{size}",
        'serial': time_serial,
        'pytorch': time_pytorch,
        'cupy': time_gpu,
        'speedup_pytorch': speedup_pytorch,
        'speedup_cupy': speedup_cupy,
        'cpu_eff': cpu_eff,
        'gpu_eff_sm': gpu_eff_sm,
        'cpu_gpu_ratio': cpu_gpu_ratio,
        'blurred_serial': blur_serial,
        'blurred_pytorch': blur_pytorch,
        'blurred_gpu': blur_gpu
    })
```

```
GPU_CORES = 2560
print("="*90)
print("Gaussian Blur ")
print(f"Image Size: {width} x {height}")
print("="*90)
print("Processing Units Used")
print(f"CPU Cores: {CPU_CORES}")
print(f"GPU: NVIDIA Tesla T4")
print(f"GPU Streaming Multiprocessors (SMs): 40")
print(f"Estimated CUDA Cores: {GPU_CORES}")
print(f"GPU Memory: 16.0 GB")
print("="*135)
print(f"{'Kernel Size':<15}{'Serial(s)':<15}{'PyTorch(s)':<15}{'CuPy(s)':<15}"
    f"{'CPU SpdUp':<15}{'GPU SpdUp':<15}{'CPU Eff':<15}{'GPU Eff(SM)':<15}{'CPU:GPU Ratio':<15}")
print("-" * 135)
for r in results:
   print(f"{r['kernel_size']:<15}{r['serial']:<15.6f}{r['pytorch']:<15.6f}"
       f"{r['cupy']:<15.6f}{r['speedup_pytorch']:<15.2f}"
       f"{r['speedup_cupy']:<15.2f}{r['cpu_eff']:<15.4f}"
       f"{r['gpu_eff_sm']:<15.4f}{r['cpu_gpu_ratio']:<15.4f}")
plt.figure(figsize=(15, 5))
last_result = results[-1]
plt.subplot(1, 4, 1); plt.imshow(image_rgb); plt.title("Original"); plt.axis('off')
plt.subplot(1, 4, 2); plt.imshow(last_result['blurred_serial']); plt.title(f"Serial
{last_result['kernel_size']}"); plt.axis('off')
plt.subplot(1, 4, 3); plt.imshow(last_result['blurred_pytorch']); plt.title("PyTorch CPU"); plt.axis('off')
plt.subplot(1, 4, 4); plt.imshow(last_result['blurred_gpu']); plt.title("CuPy GPU"); plt.axis('off')
plt.tight_layout()
plt.show()
```

OUTPUT:

IMAGE SIZE:512*512

```
Saving 512_512.jpg to 512_512 (1).jpg
================================================================================
Gaussian Blur
Image Size: 512 x 512
================================================================================
Processing Units Used
CPU Cores: 2
GPU: NVIDIA Tesla T4
GPU Streaming Multiprocessors (SMs): 40
Estimated CUDA Cores: 2560
GPU Memory: 16.0 GB
================================================================================================================================================
Kernel Size    Serial(s)      PyTorch(s)     CuPy(s)        CPU SpdUp      GPU SpdUp      CPU Eff        GPU Eff(SM)    CPU:GPU Ratio
------------------------------------------------------------------------------------------------------------------------------------------------
7x7            0.120717       0.029979       0.003215       4.03           37.55          2.0134         0.9387         9.3252
13x13          0.336201       0.080330       0.004236       4.19           79.36          2.0926         1.9840         18.9615
15x15          0.445947       0.102813       0.004609       4.34           96.75          2.1687         2.4188         22.3064
17x17          0.556859       0.141544       0.005315       3.93           104.77         1.9671         2.6191         26.6295
31x31          1.859679       0.421268       0.010517       4.41           176.82         2.2072         4.4205         40.0545
```

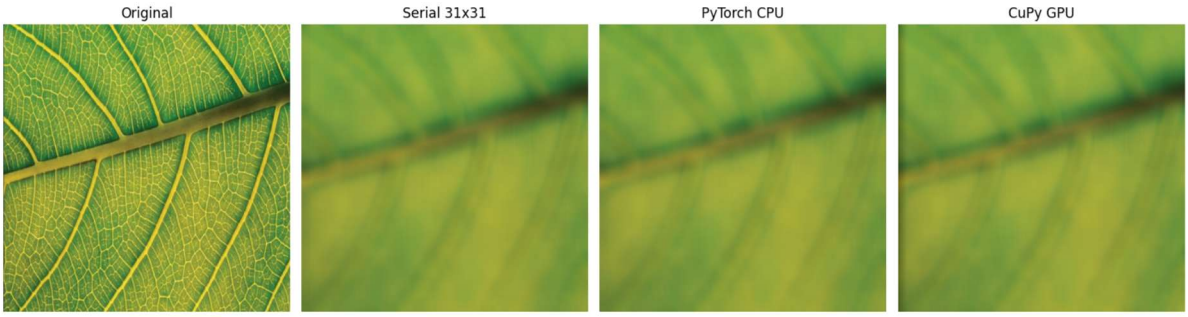


IMAGE SIZE 1024*1024

```
Saving 1024_1024.jpg to 1024_1024.jpg
================================================================================
Gaussian Blur
Image Size: 1024 x 1024
================================================================================
Processing Units Used
CPU Cores: 2
GPU: NVIDIA Tesla T4
GPU Streaming Multiprocessors (SMs): 40
Estimated CUDA Cores: 2560
GPU Memory: 16.0 GB
================================================================================================================================================
Kernel Size    Serial(s)      PyTorch(s)     CuPy(s)        CPU SpdUp      GPU SpdUp      CPU Eff        GPU Eff(SM)    CPU:GPU Ratio
------------------------------------------------------------------------------------------------------------------------------------------------
7x7            0.483783       0.156890       0.012581       3.08           38.45          1.5418         0.9613         12.4703
13x13          1.332022       0.365030       0.016687       3.65           79.82          1.8245         1.9956         21.8749
15x15          2.051415       0.877343       0.018711       2.34           109.64         1.1691         2.7410         46.8901
17x17          2.194768       0.582384       0.020130       3.77           109.03         1.8843         2.7258         28.9313
31x31          7.938442       1.763054       0.040080       4.50           198.07         2.2513         4.9517         43.9888
```

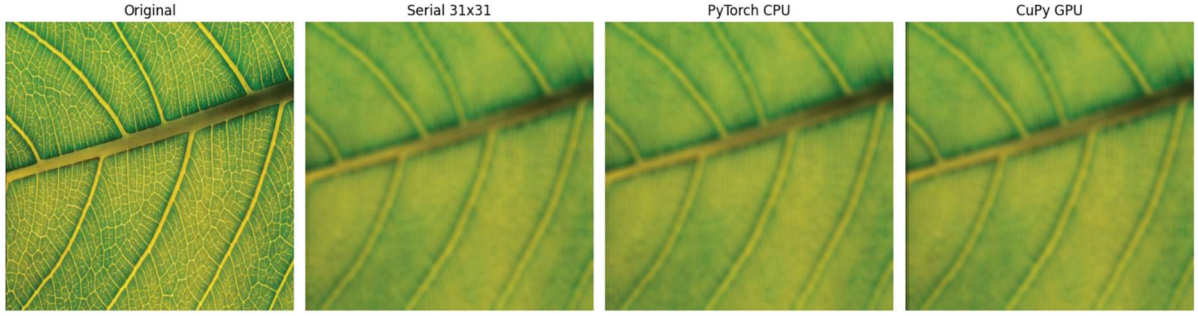
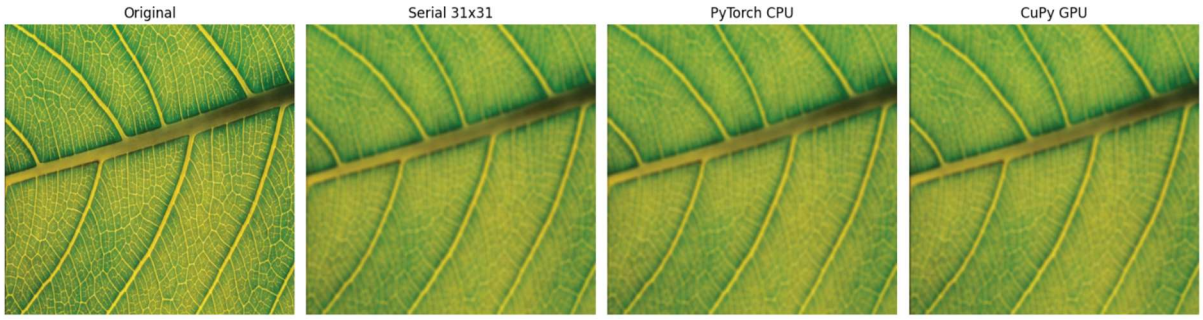


IMAGE SIZE:2048*2048

```
Saving 2048_2048.jpg to 2048_2048.jpg
================================================================================
Gaussian Blur
Image Size: 2048 x 2048
================================================================================
Processing Units Used
CPU Cores: 2
GPU: NVIDIA Tesla T4
GPU Streaming Multiprocessors (SMs): 40
Estimated CUDA Cores: 2560
GPU Memory: 16.0 GB
================================================================================================================================
Kernel Size    Serial(s)      PyTorch(s)     CuPy(s)        CPU SpdUp      GPU SpdUp      CPU Eff        GPU Eff(SM)    CPU:GPU Ratio
--------------------------------------------------------------------------------------------------------------------------------
7x7            2.112919       1.146375       0.061206       1.84           34.52          0.9216         0.8630         18.7299
13x13          5.421342       1.474091       0.070861       3.68           76.51          1.8389         1.9127         20.8027
15x15          7.790961       1.871280       0.076100       4.16           102.38         2.0817         2.5594         24.5896
17x17          9.760978       2.279889       0.092787       4.28           105.20         2.1407         2.6299         24.5713
31x31          31.496253      6.959003       0.168769       4.53           186.62         2.2630         4.6656         41.2339
```



COMPARISON TABLE:

USING 31*31 KERNEL

Execution time

S.No	Image Size	SERIAL CPU(s)	PARALLEL CPU(s)	CUDA GPU(s)
1	512*512	1.859679	0.421268	0.010517
2	1024*1024	7.938442	1.763054	0.040080
3	2048*2048	31.496253	6.959003	0.168769

Speedup

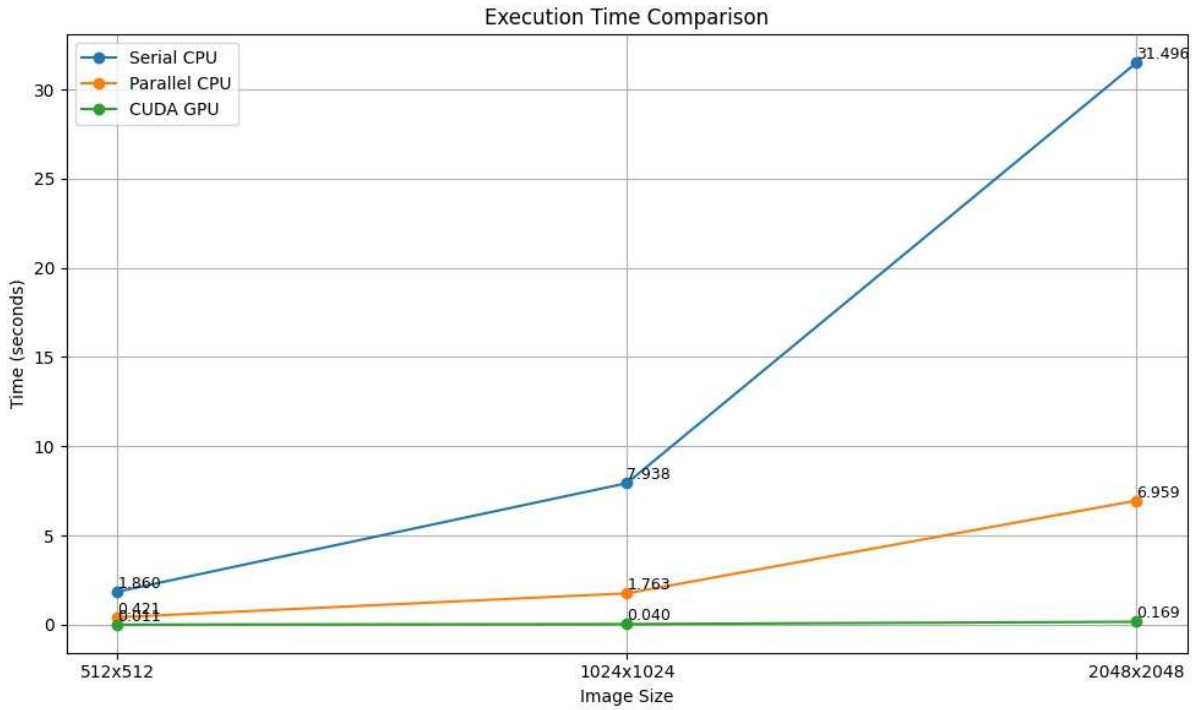
S.No	Image Size	SERIAL vs CPU	SERIAL vs GPU
1	512*512	4.41x	176.82x
2	1024*1024	4.50x	198.07x
3	2048*2048	4.53	186.62

Efficiency

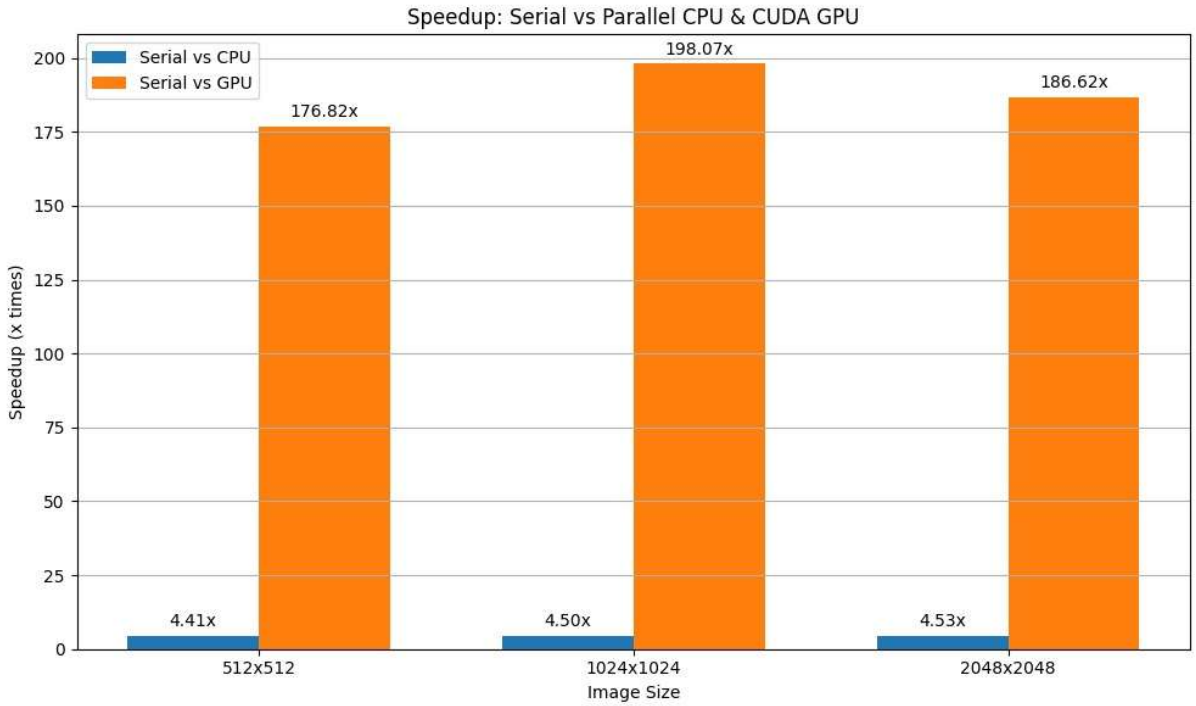
S.No	Image Size	CPU EFFICIENCY	GPU EFFICIENCY
1	512*512	2.2072	4.4205
2	1024*1024	2.2513	4.9517
3	2048*2048	2.2630	4.6656

GRAPH:

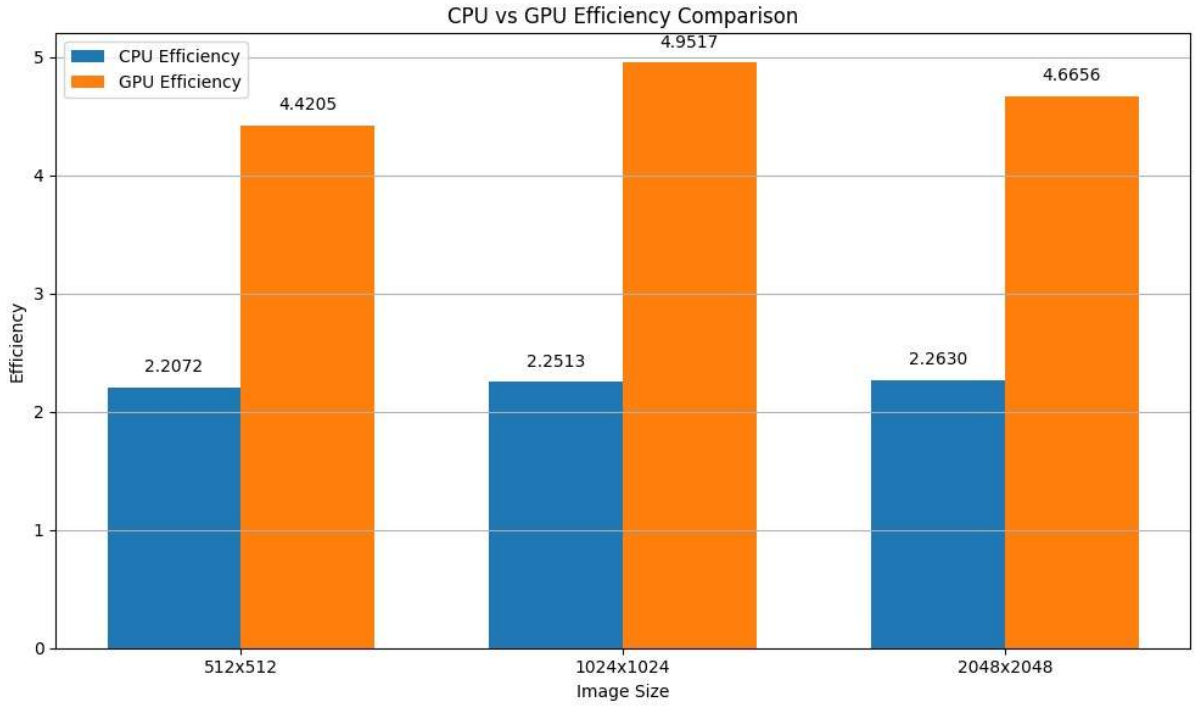
Execution time vs Image size



Speedup vs Image size



Efficiency vs Image size



2.SOBEL EDGE DETECTION ALGORITHM

CODE:

```
import cv2
import numpy as np
import torch
import cupy as cp
import time
import matplotlib.pyplot as plt
from scipy.signal import convolve2d
from google.colab import files
uploaded = files.upload()
image_path = next(iter(uploaded))
image = cv2.imread(image_path)
image_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
height, width = image_gray.shape
CPU_CORES = 2
torch.set_num_threads(CPU_CORES)
def sobel_serial(image):
    Kx = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
    Ky = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]])
    Gx = convolve2d(image, Kx, mode='same', boundary='symm')
    Gy = convolve2d(image, Ky, mode='same', boundary='symm')
    edge = np.hypot(Gx, Gy)
    return np.clip(edge, 0, 255).astype(np.uint8)
def sobel_pytorch_cpu(image):
    image_tensor = torch.tensor(image, dtype=torch.float32).unsqueeze(0).unsqueeze(0)
    Kx = torch.tensor([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], dtype=torch.float32).unsqueeze(0).unsqueeze(0)
    Ky = torch.tensor([[1, 2, 1], [0, 0, 0], [-1, -2, -1]], dtype=torch.float32).unsqueeze(0).unsqueeze(0)
    padded = torch.nn.functional.pad(image_tensor, (1, 1, 1, 1), mode='reflect')
    Gx = torch.nn.functional.conv2d(padded, Kx)
    Gy = torch.nn.functional.conv2d(padded, Ky)
    edge = torch.sqrt(Gx ** 2 + Gy ** 2).squeeze().numpy()
    return np.clip(edge, 0, 255).astype(np.uint8)
raw_sobel_code = r'''
extern "C" _global_
void sobel_edge(const float* img, float* out, int width, int height) {
    int x = blockDim.x * blockIdx.x + threadIdx.x;
    int y = blockDim.y * blockIdx.y + threadIdx.y;

    float Kx[3][3] = {{-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1}};
    float Ky[3][3] = {{1, 2, 1}, {0, 0, 0}, {-1, -2, -1}};

    if (x >= 1 && x < width - 1 && y >= 1 && y < height - 1) {
        float gx = 0.0f;
        float gy = 0.0f;
        for (int ky = -1; ky <= 1; ky++) {
```

```
            for (int kx = -1; kx <= 1; kx++) {
                float pixel = img[(y + ky) * width + (x + kx)];
                gx += pixel * Kx[ky + 1][kx + 1];
                gy += pixel * Ky[ky + 1][kx + 1];
            }
        }
        float val = sqrtf(gx * gx + gy * gy);
        out[y * width + x] = val;
    }
}
'''
sobel_module = cp.RawModule(code=raw_sobel_code)
sobel_kernel = sobel_module.get_function("sobel_edge")

def sobel_cupy(image):
    h, w = image.shape
    img_cp = cp.asarray(image, dtype=cp.float32)
    out_cp = cp.zeros_like(img_cp)
    block = (16, 16)
    grid = ((w + block[0] - 1) // block[0], (h + block[1] - 1) // block[1])
    sobel_kernel(grid, block, (img_cp, out_cp, np.int32(w), np.int32(h)))
    cp.cuda.Device(0).synchronize()
    cp._default_memory_pool.free_all_blocks()
    return cp.asnumpy(cp.clip(out_cp, 0, 255)).astype(np.uint8)
_ = sobel_pytorch_cpu(image_gray)
_ = sobel_cupy(image_gray)
start = time.time()
edge_serial = sobel_serial(image_gray)
t_serial = time.time() - start

start = time.time()
edge_pytorch = sobel_pytorch_cpu(image_gray)
t_pytorch = time.time() - start

start = time.time()
edge_gpu = sobel_cupy(image_gray)
t_gpu = time.time() - start

speedup_pytorch = t_serial / t_pytorch
speedup_cupy = t_serial / t_gpu

# Report
print("="*80)
print("Sobel Edge Detection Benchmark")
print(f"Image Size: {width} x {height}")
print(f"CPU Cores Used: {CPU_CORES}")
print(f"GPU: NVIDIA Tesla T4 (40 SMs)")
print("="*80)
```

```
print(f"{'Method':<20}{'Time (s)':<15}{'Speedup':<15}")
print("-"*50)
print(f"{'Serial CPU':<20}{t_serial:<15.6f}{'1.00':<15}")
print(f"{'PyTorch CPU':<20}{t_pytorch:<15.6f}{speedup_pytorch:<15.2f}")
print(f"{'CuPy GPU':<20}{t_gpu:<15.6f}{speedup_cupy:<15.2f}")
plt.figure(figsize=(15, 4))
plt.subplot(1, 4, 1); plt.imshow(image_gray, cmap='gray'); plt.title("Original"); plt.axis('off')
plt.subplot(1, 4, 2); plt.imshow(edge_serial, cmap='gray'); plt.title("Serial CPU"); plt.axis('off')
plt.subplot(1, 4, 3); plt.imshow(edge_pytorch, cmap='gray'); plt.title("PyTorch CPU"); plt.axis('off')
plt.subplot(1, 4, 4); plt.imshow(edge_gpu, cmap='gray'); plt.title("CuPy GPU"); plt.axis('off')
plt.tight_layout()
plt.show()
```

OUTPUT:

IMAGE SIZE:512*512

```
Saving 512_512.jpg to 512_512 (5).jpg
================================================================================
Sobel Edge Detection Benchmark
Image Size: 512 x 512
CPU Cores Used: 2
GPU: NVIDIA Tesla T4 (40 SMs)
================================================================================
Method               Time (s)        Speedup
--------------------------------------------------
Serial CPU           0.029054        1.00
PyTorch CPU          0.006484        4.48
CuPy GPU             0.001242        23.40
```

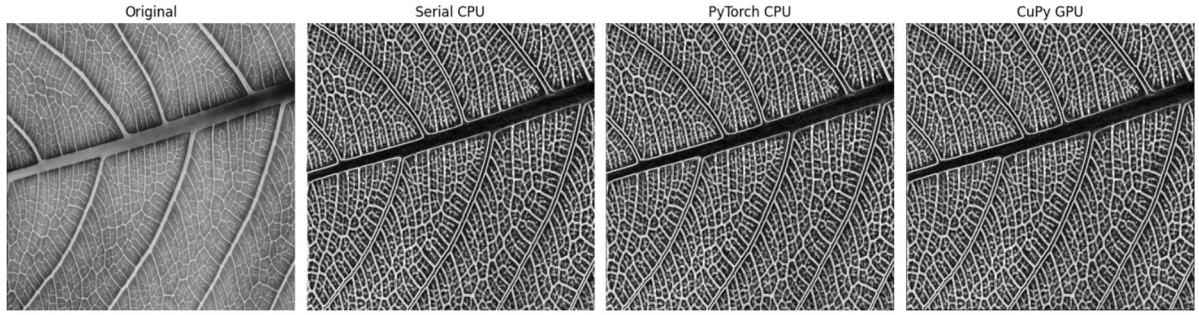


IMAGE SIZE 1024*1024

```
================================================================================
Sobel Edge Detection Benchmark
Image Size: 1024 x 1024
CPU Cores Used: 2
GPU: NVIDIA Tesla T4 (40 SMs)
================================================================================
Method               Time (s)       Speedup
------------------------------------------------------
Serial CPU           0.117111       1.00
PyTorch CPU          0.090639       1.29
CuPy GPU             0.003656       32.03
```

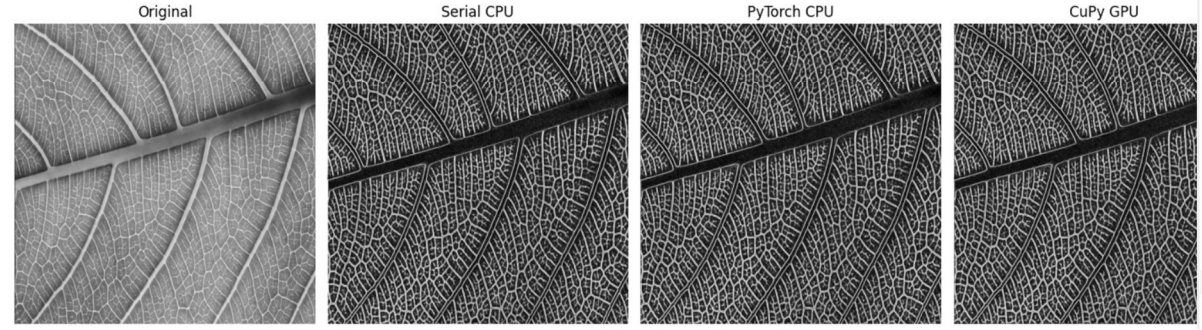


IMAGE SIZE:2048*2048

```
================================================================================
Sobel Edge Detection Benchmark
Image Size: 2048 x 2048
CPU Cores Used: 2
GPU: NVIDIA Tesla T4 (40 SMs)
================================================================================
Method               Time (s)       Speedup
------------------------------------------------------
Serial CPU           0.490542       1.00
PyTorch CPU          0.490549       1.00
CuPy GPU             0.013215       37.12
```

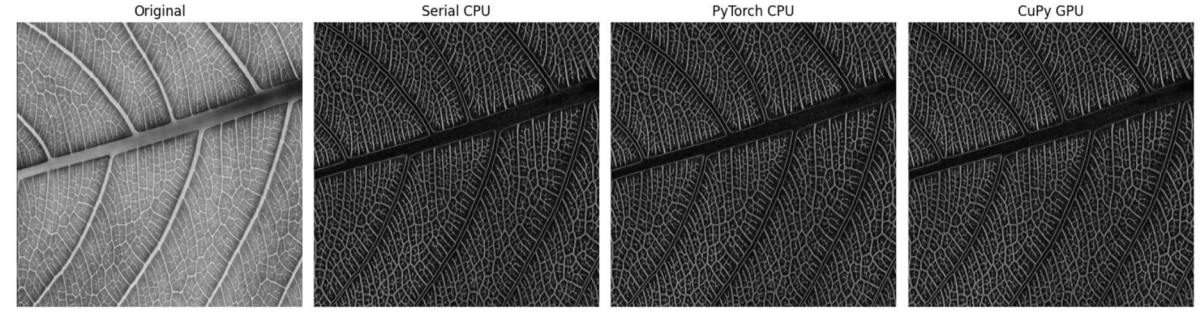
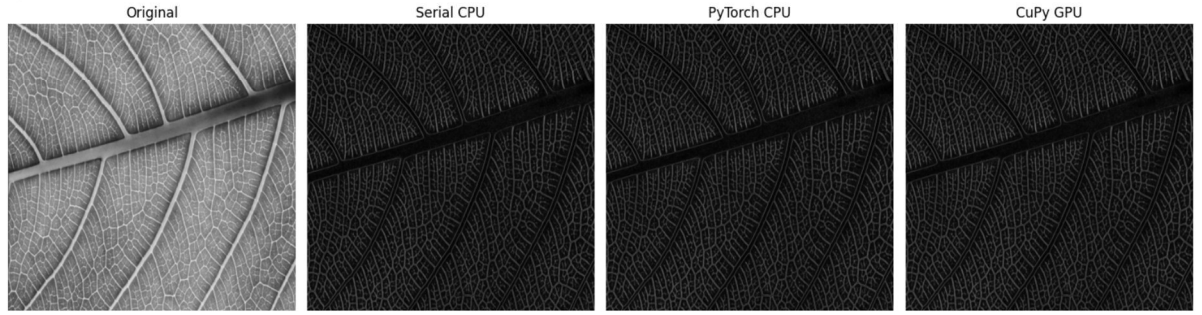


IMAGE SIZE:4000*4000

```
--------------------------------------------------------------------------------
Sobel Edge Detection Benchmark
Image Size: 4000 x 4000
CPU Cores Used: 2
GPU: NVIDIA Tesla T4 (40 SMs)
================================================================================
Method               Time (s)       Speedup
------------------------------------------------------
Serial CPU           1.995513       1.00
PyTorch CPU          2.275235       0.88
CuPy GPU             0.067154       29.72
```



COMPARISON TABLE:

Execution time

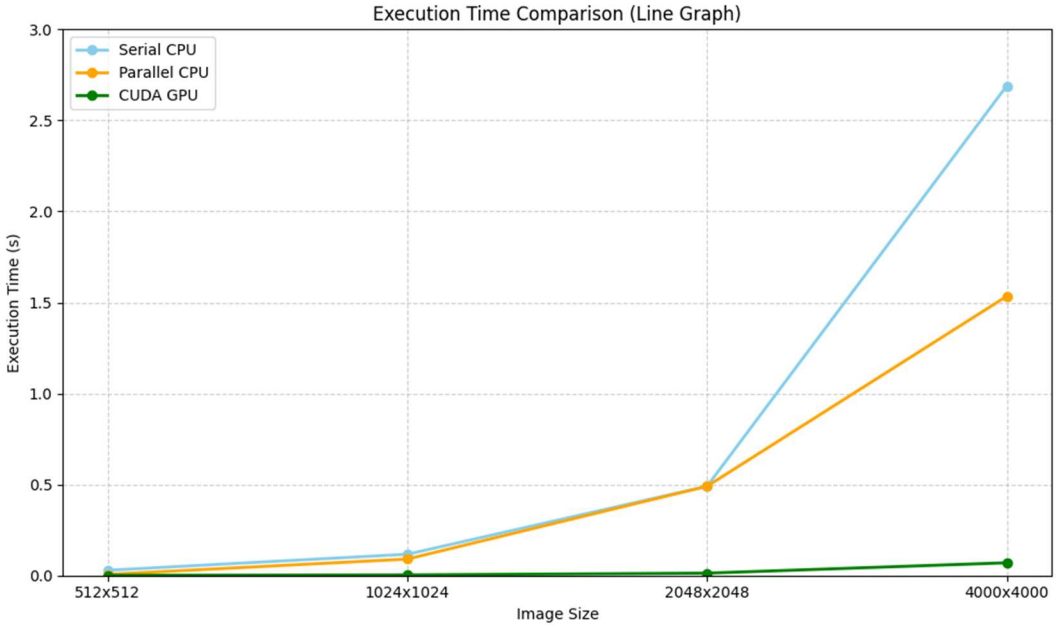
S.No	Image size	SERIAL CPU(s)	PARALLEL CPU(s)	CUDA GPU(s)
1	512*512	0.029054	0.006484	0.001242
2	1024*1024	0.117111	0.090639	0.003656
3	2048*2048	0.490542	0.490549	0.013215
4	4000*4000	2.689592	1.534121	0.070198

Speedup

S.No	Image Size	SERIAL vs CPU	SERIAL vs GPU
1	512*512	4.48x	23.40x
2	1024*1024	1.29x	32.03x
3	2048*2048	1.00x	37.12x
4	4000*4000	1.75x	38.31x

GRAPH:

Execution time vs Image size



Speedup vs Image size

3.PREWITT EDGE DETECTION ALGORITHM

CODE:

```
import cv2
import numpy as np
import torch
import cupy as cp
import time
import matplotlib.pyplot as plt
from scipy.signal import convolve2d
from google.colab import files

uploaded = files.upload()
image_path = next(iter(uploaded))
image = cv2.imread(image_path)
image_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
height, width = image_gray.shape

CPU_CORES = 2
torch.set_num_threads(CPU_CORES)


def prewitt_serial(image):
    Kx = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])
    Ky = np.array([[1, 1, 1], [0, 0, 0], [-1, -1, -1]])
    Gx = convolve2d(image, Kx, mode='same', boundary='symm')
    Gy = convolve2d(image, Ky, mode='same', boundary='symm')
    edge = np.hypot(Gx, Gy)
    return np.clip(edge, 0, 255).astype(np.uint8)


def prewitt_pytorch_cpu(image):
    image_tensor = torch.tensor(image, dtype=torch.float32).unsqueeze(0).unsqueeze(0)
    Kx = torch.tensor([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]], dtype=torch.float32).unsqueeze(0).unsqueeze(0)
    Ky = torch.tensor([[1, 1, 1], [0, 0, 0], [-1, -1, -1]], dtype=torch.float32).unsqueeze(0).unsqueeze(0)
    padded = torch.nn.functional.pad(image_tensor, (1, 1, 1, 1), mode='reflect')
    Gx = torch.nn.functional.conv2d(padded, Kx)
    Gy = torch.nn.functional.conv2d(padded, Ky)
    edge = torch.sqrt(Gx ** 2 + Gy ** 2).squeeze().numpy()
    return np.clip(edge, 0, 255).astype(np.uint8)


raw_prewitt_code = r'''
extern "C" _global_
void prewitt_edge(const float* img, float* out, int width, int height) {
    int x = blockDim.x * blockIdx.x + threadIdx.x;
    int y = blockDim.y * blockIdx.y + threadIdx.y;
```

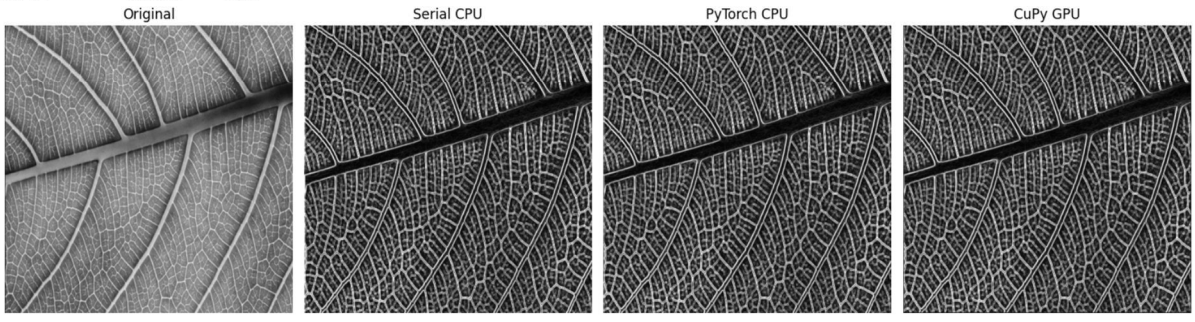
```
    float Kx[3][3] = {{-1, 0, 1}, {-1, 0, 1}, {-1, 0, 1}};
    float Ky[3][3] = {{1, 1, 1}, {0, 0, 0}, {-1, -1, -1}};

    if (x >= 1 && x < width - 1 && y >= 1 && y < height - 1) {
        float gx = 0.0f;
        float gy = 0.0f;
        for (int ky = -1; ky <= 1; ky++) {
            for (int kx = -1; kx <= 1; kx++) {
                float pixel = img[(y + ky) * width + (x + kx)];
                gx += pixel * Kx[ky + 1][kx + 1];
                gy += pixel * Ky[ky + 1][kx + 1];
            }
        }
        float val = sqrtf(gx * gx + gy * gy);
        out[y * width + x] = val;
    }
}
'''
prewitt_module = cp.RawModule(code=raw_prewitt_code)
prewitt_kernel = prewitt_module.get_function("prewitt_edge")

def prewitt_cupy(image):
    h, w = image.shape
    img_cp = cp.asarray(image, dtype=cp.float32)
    out_cp = cp.zeros_like(img_cp)
    block = (16, 16)
    grid = ((w + block[0] - 1) // block[0], (h + block[1] - 1) // block[1])
    prewitt_kernel(grid, block, (img_cp, out_cp, np.int32(w), np.int32(h)))
    cp.cuda.Device(0).synchronize()
    cp._default_memory_pool.free_all_blocks()
    return cp.asnumpy(cp.clip(out_cp, 0, 255)).astype(np.uint8)


_ = prewitt_pytorch_cpu(image_gray)
_ = prewitt_cupy(image_gray)
start = time.time()
edge_serial = prewitt_serial(image_gray)
t_serial = time.time() - start

start = time.time()
edge_pytorch = prewitt_pytorch_cpu(image_gray)
t_pytorch = time.time() - start

start = time.time()
edge_gpu = prewitt_cupy(image_gray)
t_gpu = time.time() - start
```

```
speedup_pytorch = t_serial / t_pytorch
speedup_cupy = t_serial / t_gpu
print("="*80)
print("Prewitt Edge Detection Benchmark")
print(f"Image Size: {width} x {height}")
print(f"CPU Cores Used: {CPU_CORES}")
print(f"GPU: NVIDIA Tesla T4 (40 SMs)")
print("="*80)
print(f"{'Method':<20}{'Time (s)':<15}{'Speedup':<15}")
print("-"*50)
print(f"{'Serial CPU':<20}{t_serial:<15.6f}")
print(f"{'PyTorch CPU':<20}{t_pytorch:<15.6f}{speedup_pytorch:<15.2f}")
print(f"{'CuPy GPU':<20}{t_gpu:<15.6f}{speedup_cupy:<15.2f}")
plt.figure(figsize=(15, 4))
plt.subplot(1, 4, 1); plt.imshow(image_gray, cmap='gray'); plt.title("Original"); plt.axis('off')
plt.subplot(1, 4, 2); plt.imshow(edge_serial, cmap='gray'); plt.title("Serial CPU"); plt.axis('off')
plt.subplot(1, 4, 3); plt.imshow(edge_pytorch, cmap='gray'); plt.title("PyTorch CPU"); plt.axis('off')
plt.subplot(1, 4, 4); plt.imshow(edge_gpu, cmap='gray'); plt.title("CuPy GPU"); plt.axis('off')
plt.tight_layout()
plt.show()
```

OUTPUT:

IMAGE SIZE:512*512

```
================================================================================
Prewitt Edge Detection Benchmark
Image Size: 512 x 512
CPU Cores Used: 2
GPU: NVIDIA Tesla T4 (40 SMs)
================================================================================
Method               Time (s)       Speedup
--------------------------------------------------
Serial CPU           0.041506
PyTorch CPU          0.013796       3.01
CuPy GPU             0.002023       20.52
```



I

MAGE SIZE 1024*1024

```
================================================================================
Prewitt Edge Detection Benchmark
Image Size: 1024 x 1024
CPU Cores Used: 2
GPU: NVIDIA Tesla T4 (40 SMs)
================================================================================
Method                Time (s)        Speedup
--------------------------------------------------
Serial CPU            0.112440
PyTorch CPU           0.084323        1.33
CuPy GPU              0.003691        30.46
```

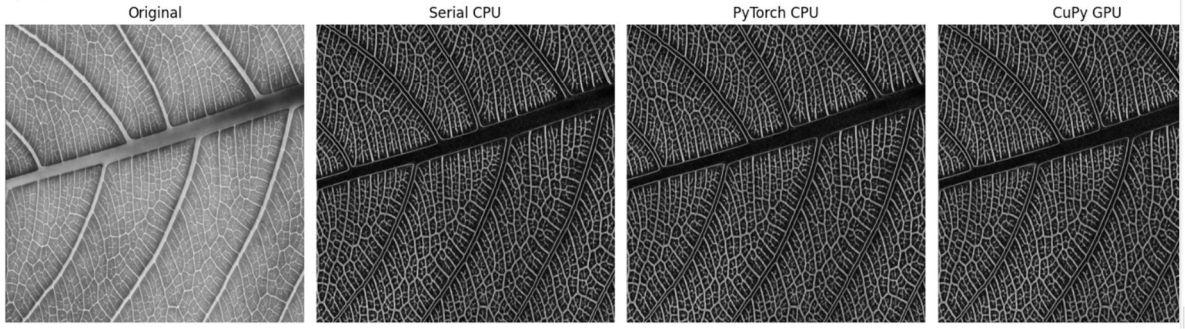


IMAGE SIZE:2048*2048

```
Prewitt Edge Detection Benchmark
Image Size: 2048 x 2048
CPU Cores Used: 2
GPU: NVIDIA Tesla T4 (40 SMs)
================================================================================
Method                Time (s)        Speedup
--------------------------------------------------
Serial CPU            0.480191
PyTorch CPU           0.355720        1.35
CuPy GPU              0.012436        38.61
```

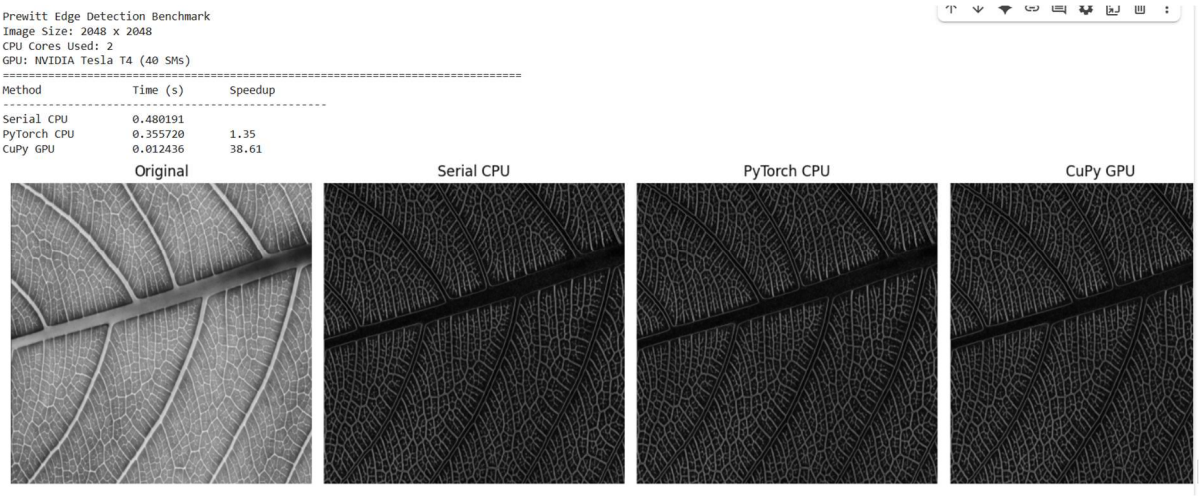
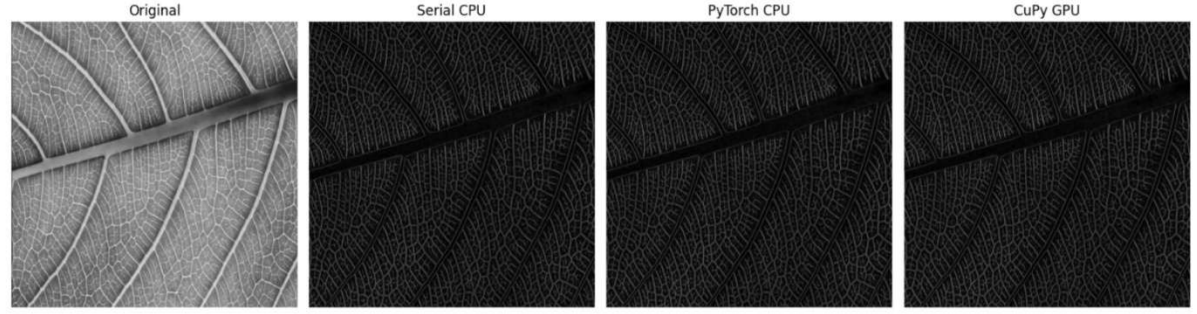


IMAGE SIZE:4000*4000

```
================================================================================
Sobel Edge Detection Benchmark
Image Size: 4000 x 4000
CPU Cores Used: 2
GPU: NVIDIA Tesla T4 (40 SMs)
================================================================================
Method                Time (s)        Speedup
--------------------------------------------------
Serial CPU            2.689592        1.00
PyTorch CPU           1.534121        1.75
CuPy GPU              0.070198        38.31
```



COMPARISON TABLE:

Execution time

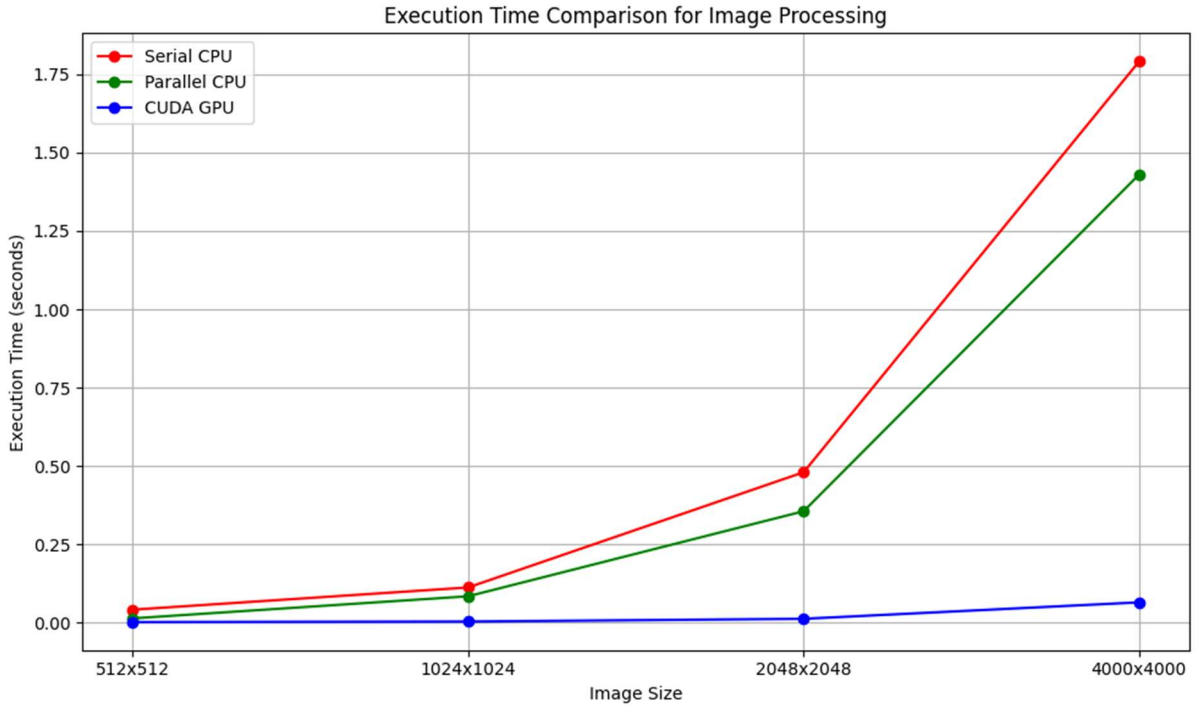
S.No	Image size	SERIAL CPU(s)	PARALLEL CPU(s)	CUDA GPU(s)
1	512*512	0.041506	0.013796	0.002023
2	1024*1024	0.112440	0.084323	0.003691
3	2048*2048	0.480191	0.355720	0.012436
4	4000*4000	1.791344	1.430575	0.064933

Speedup

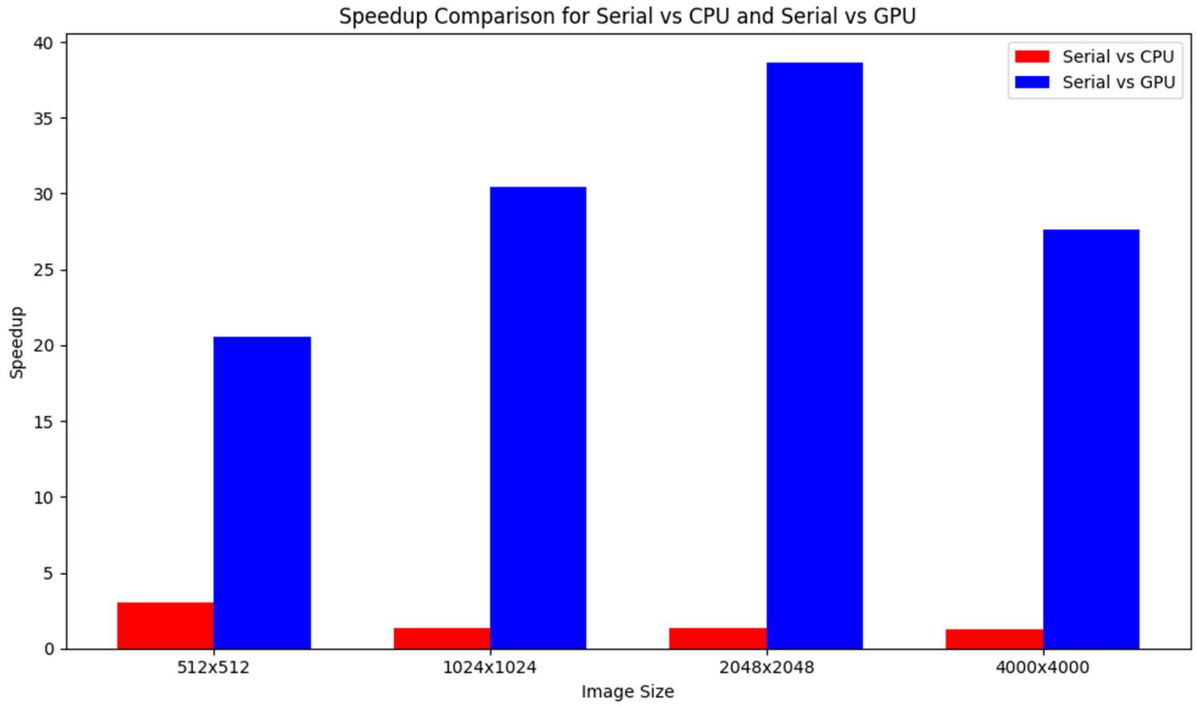
S.No	Image Size	SERIAL vs CPU	SERIAL vs GPU
1	512*512	3.01x	20.52x
2	1024*1024	1.33x	30.46x
3	2048*2048	1.35x	38.61x
4	4000*4000	1.25x	27.59x

GRAPH:

Execution time vs Image size



Speedup vs Image size



4.ROBERTS EDGE DETECTION ALGORITHM

CODE:

```
import cv2
import numpy as np
import torch
import cupy as cp
import time
import matplotlib.pyplot as plt
from scipy.signal import convolve2d
from google.colab import files
uploaded = files.upload()
image_path = next(iter(uploaded))
image = cv2.imread(image_path)
image_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
height, width = image_gray.shape
CPU_CORES = 2
torch.set_num_threads(CPU_CORES)
def roberts_serial(image):
    Kx = np.array([[1, 0], [0, -1]])
    Ky = np.array([[0, 1], [-1, 0]])
    Gx = convolve2d(image, Kx, mode='same', boundary='symm')
    Gy = convolve2d(image, Ky, mode='same', boundary='symm')
    edge = np.hypot(Gx, Gy)
    return np.clip(edge, 0, 255).astype(np.uint8)
def roberts_pytorch_cpu(image):
    image_tensor = torch.tensor(image, dtype=torch.float32).unsqueeze(0).unsqueeze(0)
    Kx = torch.tensor([[1, 0], [0, -1]], dtype=torch.float32).unsqueeze(0).unsqueeze(0)
    Ky = torch.tensor([[0, 1], [-1, 0]], dtype=torch.float32).unsqueeze(0).unsqueeze(0)
    padded = torch.nn.functional.pad(image_tensor, (0, 1, 0, 1), mode='reflect')
```

```
    Gx = torch.nn.functional.conv2d(padded, Kx)
    Gy = torch.nn.functional.conv2d(padded, Ky)
    edge = torch.sqrt(Gx ** 2 + Gy ** 2).squeeze().numpy()
    return np.clip(edge, 0, 255).astype(np.uint8)
raw_roberts_code = r'''
extern "C" __global__
void roberts_edge(const float* img, float* out, int width, int height) {
    int x = blockDim.x * blockIdx.x + threadIdx.x;
    int y = blockDim.y * blockIdx.y + threadIdx.y;
    if (x < width - 1 && y < height - 1) {
        float gx = img[y * width + x] - img[(y + 1) * width + (x + 1)];
        float gy = img[y * width + (x + 1)] - img[(y + 1) * width + x];
        float val = sqrtf(gx * gx + gy * gy);
        out[y * width + x] = val;
    }
}
'''
roberts_module = cp.RawModule(code=raw_roberts_code)
roberts_kernel = roberts_module.get_function("roberts_edge")
def roberts_cupy(image):
    h, w = image.shape
    img_cp = cp.asarray(image, dtype=cp.float32)
    out_cp = cp.zeros_like(img_cp)
    block = (16, 16)
    grid = ((w + block[0] - 1) // block[0], (h + block[1] - 1) // block[1])
    roberts_kernel(grid, block, (img_cp, out_cp, np.int32(w), np.int32(h)))
    cp.cuda.Device(0).synchronize()
    cp._default_memory_pool.free_all_blocks()
    return cp.asnumpy(cp.clip(out_cp, 0, 255)).astype(np.uint8)
_ = roberts_pytorch_cpu(image_gray)
_ = roberts_cupy(image_gray)
```

```
start = time.time()
edge_serial = roberts_serial(image_gray)
t_serial = time.time() - start
start = time.time()
edge_pytorch = roberts_pytorch_cpu(image_gray)
t_pytorch = time.time() - start
start = time.time()
edge_gpu = roberts_cupy(image_gray)
t_gpu = time.time() - start
speedup_pytorch = t_serial / t_pytorch
speedup_cupy = t_serial / t_gpu
print("="*80)
print("Robert Edge Detection Benchmark")
print(f"Image Size: {width} x {height}")
print(f"CPU Cores Used: {CPU_CORES}")
print(f"GPU: NVIDIA Tesla T4 (40 SMs)")
print("="*80)
print(f"{'Method':<20}{'Time (s)':<15}{'Speedup':<15}")
print("-"*50)
print(f"{'Serial CPU':<20}{t_serial:<15.6f}")
print(f"{'PyTorch CPU':<20}{t_pytorch:<15.6f}{speedup_pytorch:<15.2f}")
print(f"{'CuPy GPU':<20}{t_gpu:<15.6f}{speedup_cupy:<15.2f}")
plt.figure(figsize=(15, 4))
plt.subplot(1, 4, 1); plt.imshow(image_gray, cmap='gray'); plt.title("Original"); plt.axis('off')
plt.subplot(1, 4, 2); plt.imshow(edge_serial, cmap='gray'); plt.title("Serial CPU"); plt.axis('off')
plt.subplot(1, 4, 3); plt.imshow(edge_pytorch, cmap='gray'); plt.title("PyTorch CPU"); plt.axis('off')
plt.subplot(1, 4, 4); plt.imshow(edge_gpu, cmap='gray'); plt.title("CuPy GPU"); plt.axis('off')
plt.tight_layout()
plt.show()
```

OUTPUT:

IMAGE SIZE:512*512

```
================================================================================
Robert Edge Detection Benchmark
Image Size: 512 x 512
CPU Cores Used: 2
GPU: NVIDIA Tesla T4 (40 SMs)
================================================================================
Method               Time (s)       Speedup
--------------------------------------------------
Serial CPU           0.024258
PyTorch CPU          0.011927       2.03
CuPy GPU             0.001396       17.37
```

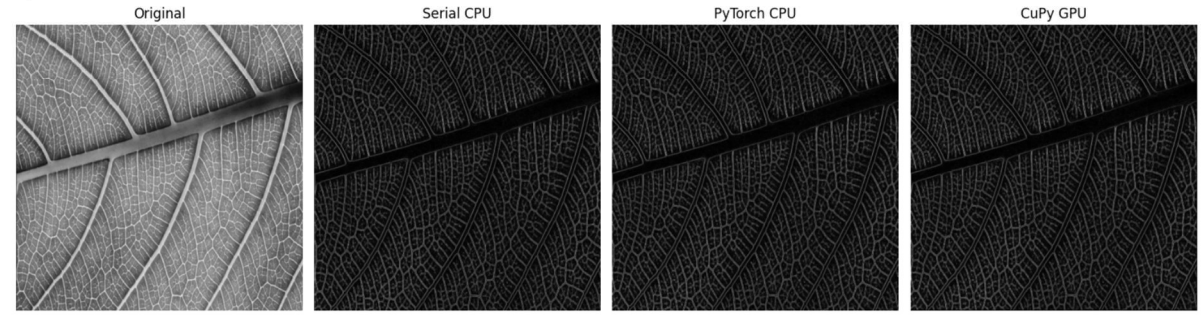


IMAGE SIZE:1024*1024

```
================================================================================
Robert Edge Detection Benchmark
Image Size: 1024 x 1024
CPU Cores Used: 2
GPU: NVIDIA Tesla T4 (40 SMs)
================================================================================
Method               Time (s)       Speedup
--------------------------------------------------
Serial CPU           0.090137       1.00
PyTorch CPU          0.083059       1.09
CuPy GPU             0.004145       21.75
```

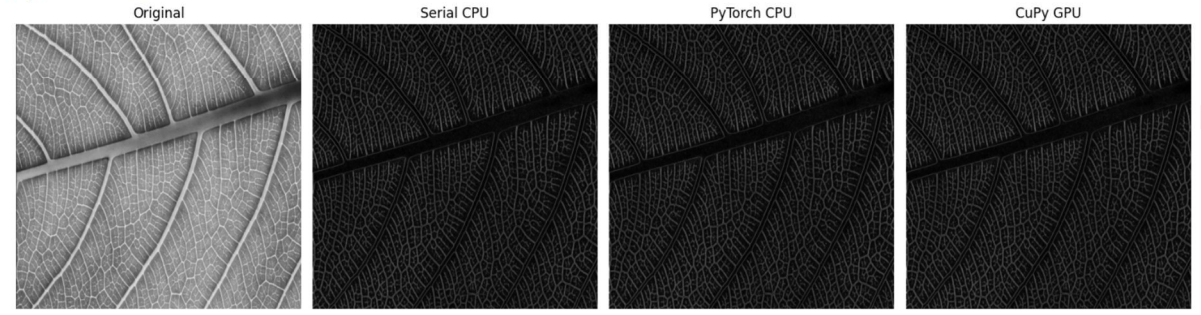
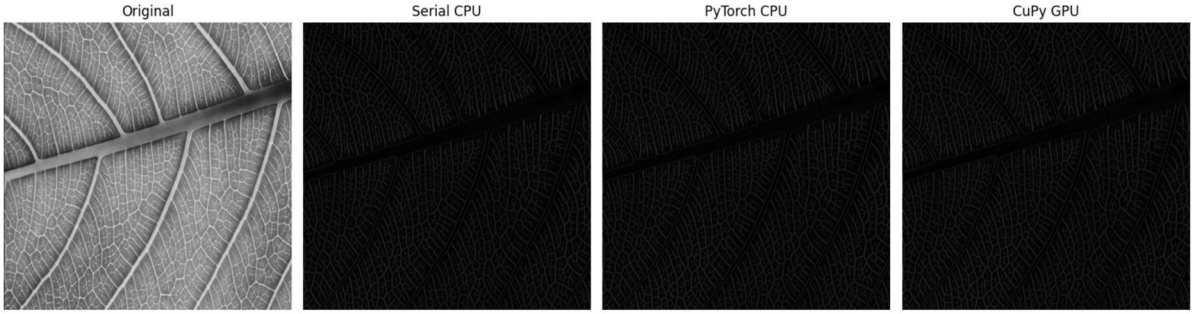


IMAGE SIZE:2048*2048

```
================================================================================
Robert Edge Detection Benchmark
Image Size: 2048 x 2048
CPU Cores Used: 2
GPU: NVIDIA Tesla T4 (40 SMs)
================================================================================
Method               Time (s)       Speedup
--------------------------------------------------
Serial CPU           0.943874
PyTorch CPU          0.873887       1.08
CuPy GPU             0.015025       62.82
```



COMPARISON TABLE:

Execution time

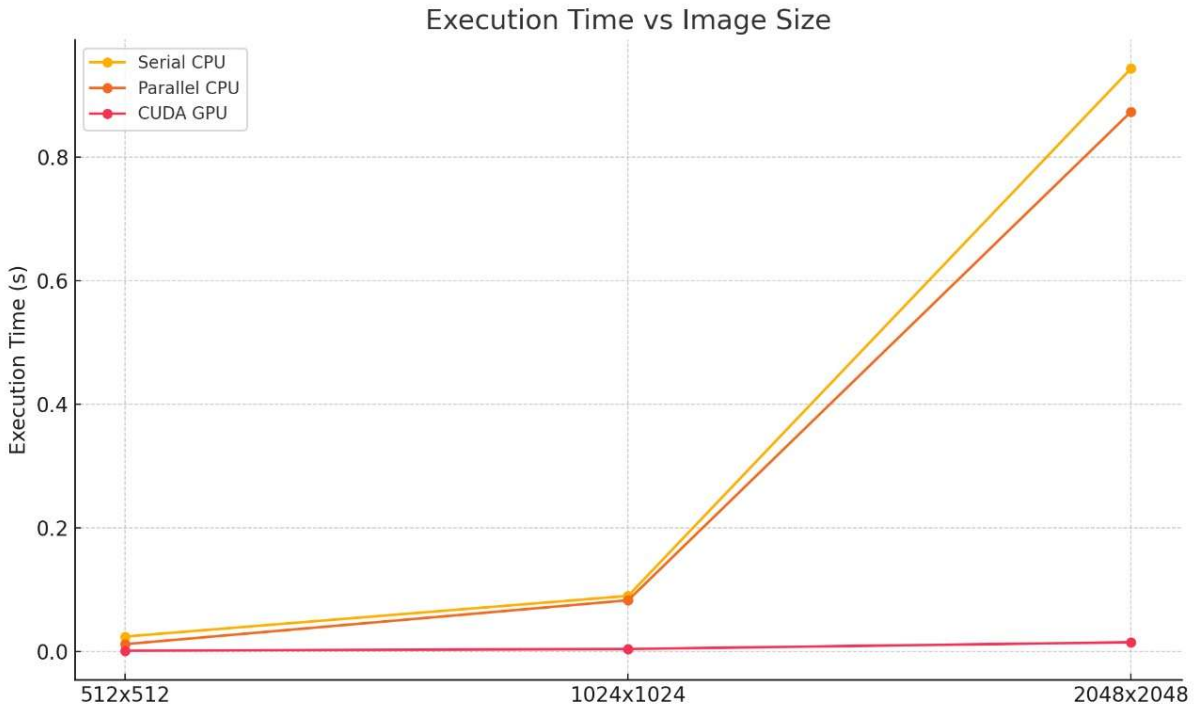
S.No	Image size	SERIAL CPU(s)	PARALLEL CPU(s)	CUDA GPU(s)
1	512*512	0.024258	0.011927	0.001396
2	1024*1024	0.090137	0.083059	0.004145
3	2048*2048	0.943874	0.873887	0.015025

Speedup

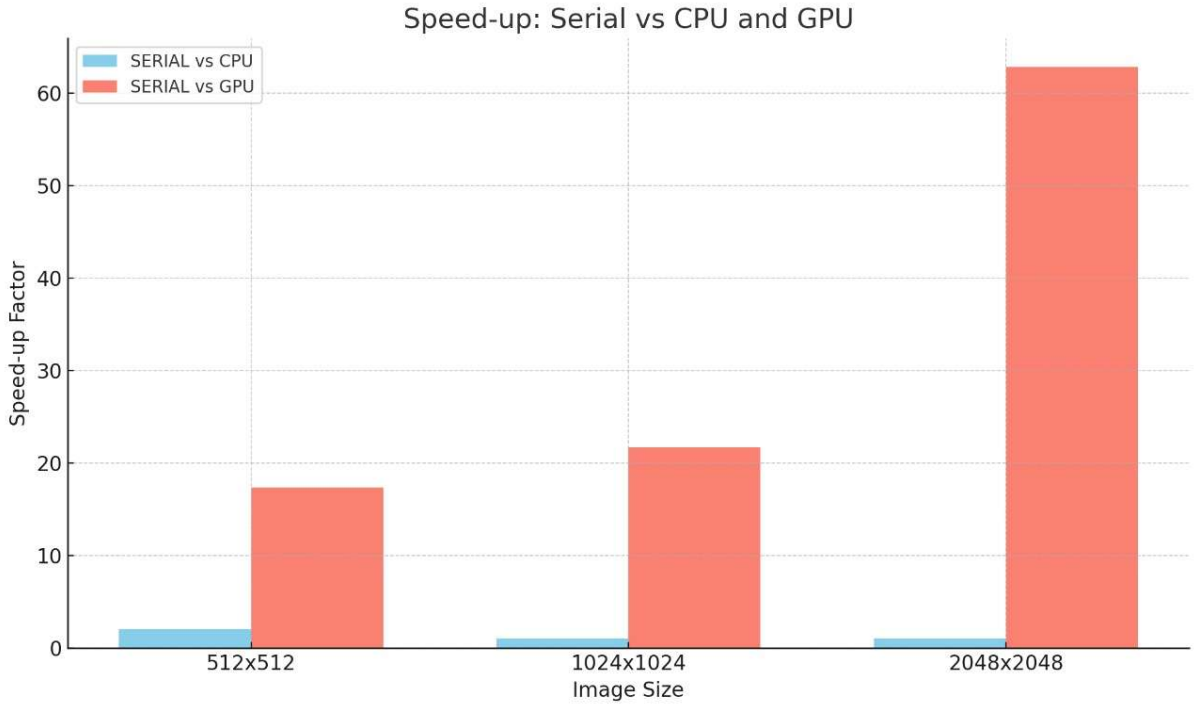
S.No	Image Size	SERIAL vs CPU	SERIAL vs GPU
1	512*512	2.03x	17.37x
2	1024*1024	1.09x	21.75x
3	2048*2048	1.08x	62.82x

GRAPH:

Execution time vs Image size



Speedup vs Image size



5.OTSU'S THRESHOLDING ALGORITHM

CODE:

```
import cv2
import numpy as np
import cupy as cp
import torch
import time
from google.colab import files
from matplotlib import pyplot as plt
uploaded = files.upload()
image_path = list(uploaded.keys())[0]
img = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
plt.imshow(img, cmap='gray')
plt.title("Original Image")
plt.axis("off")
plt.show()
def otsu_numpy(image):
    hist, _ = np.histogram(image.ravel(), 256, [0, 256])
    total = image.size
    sumB = 0.0
    wB = 0
    maximum = 0.0
    sum1 = np.dot(np.arange(256), hist)
```

```
    for t in range(256):
        wB += hist[t]
        if wB == 0:
            continue
        wF = total - wB
        if wF == 0:
            break
        sumB += t * hist[t]
        mB = sumB / wB
        mF = (sum1 - sumB) / wF
        between = wB * wF * (mB - mF) ** 2
        if between >= maximum:
            threshold = t
            maximum = between
    return threshold
def otsu_cupy(image):
    img_cp = cp.asarray(image, dtype=cp.uint8)
    hist_cp = cp.histogram(img_cp.ravel(), bins=256, range=(0, 256))[0]
    total = img_cp.size
    prob = hist_cp / total
    omega = cp.cumsum(prob)
    mu = cp.cumsum(cp.arange(256) * prob)
    mu_t = mu[-1]
    numerator = (mu_t * omega - mu) ** 2
    denominator = omega * (1.0 - omega)
    sigma_b_squared = cp.divide(numerator, denominator + 1e-10)
    threshold = int(cp.argmax(sigma_b_squared))
    return threshold
def otsu_pytorch_fast(image):
    img_tensor = torch.tensor(image, dtype=torch.float32).flatten()
    hist = torch.histc(img_tensor, bins=256, min=0, max=255)
    prob = hist / img_tensor.numel()
    levels = torch.arange(256, dtype=torch.float32)
    omega = torch.cumsum(prob, dim=0)
    mu = torch.cumsum(prob * levels, dim=0)
    mu_t = mu[-1]
    sigma_b_squared = ((mu_t * omega - mu) ** 2) / (omega * (1.0 - omega) + 1e-10)
    threshold = int(torch.argmax(sigma_b_squared))
    return threshold
def benchmark(method, name, repeat=3):
    start = time.time()
    for _ in range(repeat):
        threshold = method(img)
    duration = (time.time() - start) / repeat
    print(f"{name}: Threshold = {threshold}, Avg Time = {duration:.6f} s")
    return threshold, duration
print("Running Otsu's Threshold Benchmarks...\n")
t_np, time_np = benchmark(otsu_numpy, "NumPy Serial CPU")
```

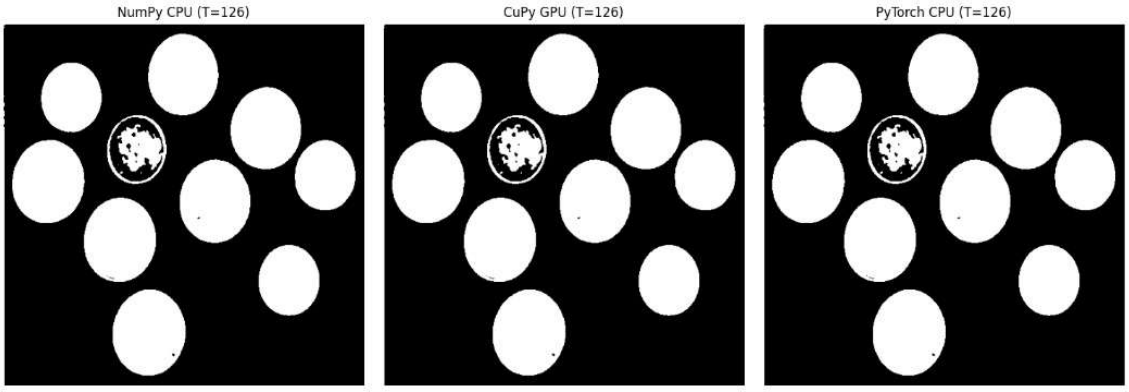
```
t_cp, time_cp = benchmark(otsu_cupy, "CuPy GPU Parallel")
t_torch, time_torch = benchmark(otsu_pytorch_fast, "PyTorch CPU")
def print_speedup(cpu_time, other_time, label):
    speedup = cpu_time / other_time
    print(f"{label} Speedup: {speedup:.2f}x")
print("\n--- Speedups ---")
print_speedup(time_np, time_cp, "CuPy GPU")
print_speedup(time_np, time_torch, "PyTorch CPU
def apply_threshold(image, threshold):
    return (image > threshold).astype(np.uint8) * 255
img_np = apply_threshold(img, t_np)
img_cp = apply_threshold(img, t_cp)
img_torch = apply_threshold(img, t_torch)
# === STEP 11: Display Output ===
plt.figure(figsize=(15, 5))
plt.subplot(1, 3, 1)
plt.imshow(img_np, cmap='gray')
plt.title(f"NumPy CPU (T={t_np})")
plt.axis("off")
plt.subplot(1, 3, 2)
plt.imshow(img_cp, cmap='gray')
plt.title(f"CuPy GPU (T={t_cp})")
plt.axis("off")
plt.subplot(1, 3, 3)
plt.imshow(img_torch, cmap='gray')
plt.title(f"PyTorch CPU (T={t_torch})")
plt.axis("off")
plt.tight_layout()
plt.show()
```

OUTPUT:

Original

IMAGE SIZE:512*512

```
NumPy Serial CPU: Threshold = 126, Avg Time = 0.005781 s
CuPy GPU Parallel: Threshold = 126, Avg Time = 0.001685 s
PyTorch CPU: Threshold = 126, Avg Time = 0.001957 s

--- Speedups ---
CuPy GPU Speedup: 3.43x
PyTorch CPU Speedup: 2.95x
```



✓ 10s completed at 7:25PM

IMAGE SIZE:1024*1024

```
Running Otsu's Threshold Benchmarks...

NumPy Serial CPU: Threshold = 126, Avg Time = 0.020068 s
CuPy GPU Parallel: Threshold = 126, Avg Time = 0.001607 s
PyTorch CPU: Threshold = 125, Avg Time = 0.014177 s

--- Speedups ---
CuPy GPU Speedup: 12.49x
PyTorch CPU Speedup: 1.42x
```

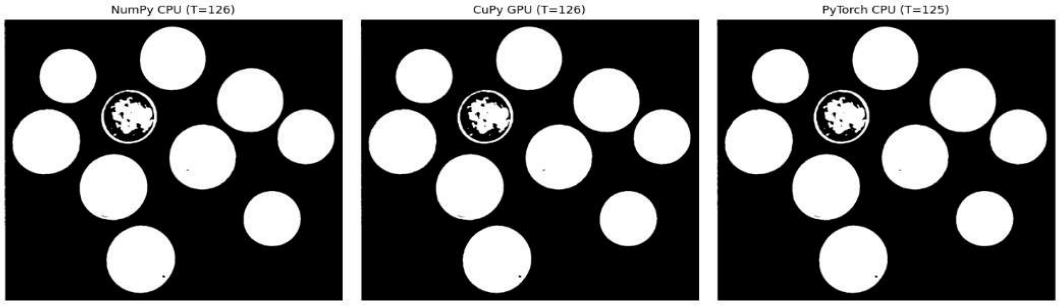


IMAGE SIZE:2048*2048

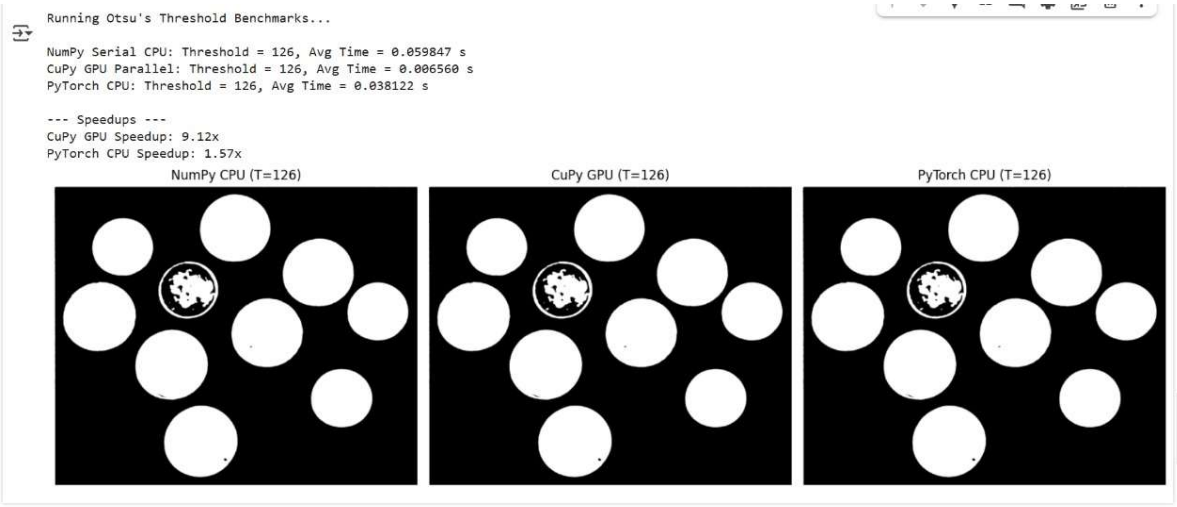
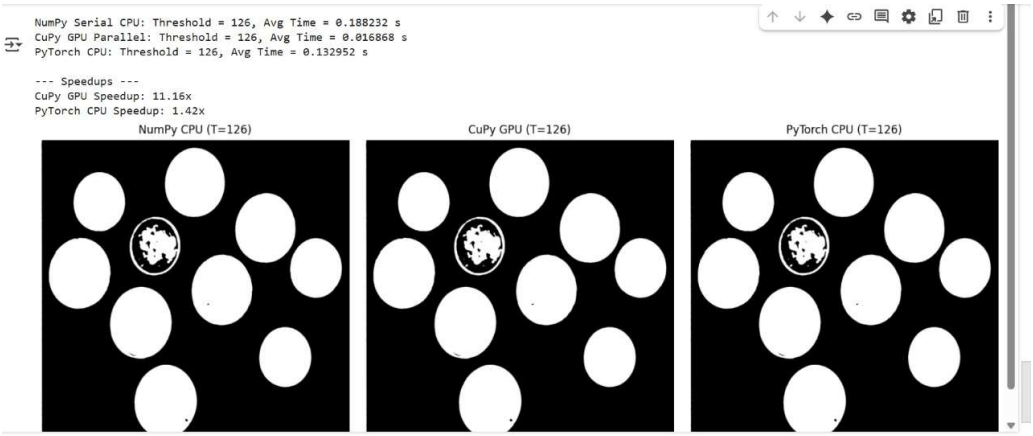


IMAGE SIZE:4000*4000



COMPARISON TABLE:

Execution Time

S.No	Image Size	SERIAL CPU(s)	PARALLEL CPU(s)	CUDA GPU(s)
1	512*512	0.005781	0.001957	0.001685
2	1024*1024	0.020068	0.014177	0.001607
3	2048*2048	0.059847	0.038122	0.006560
4	4000*4000	0.188232	0.132952	0.016868

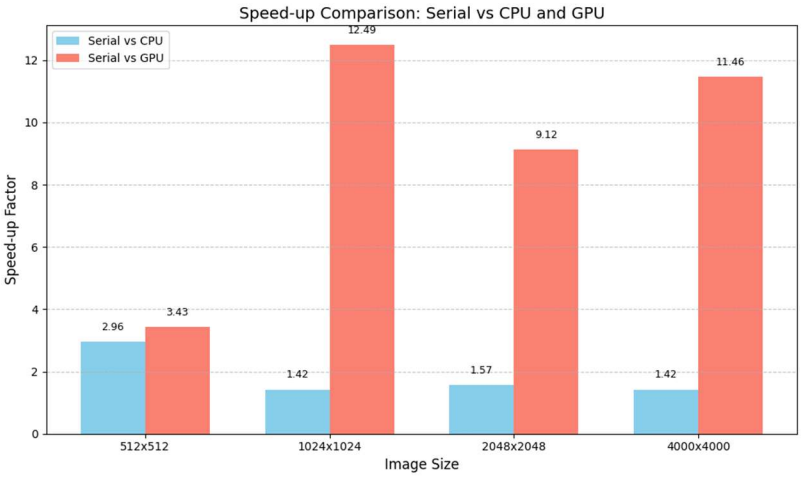
SpeedUp

S.No	Image Size	SERIAL vs CPU	SERIAL vs GPU
1	512*512	2.96x	3.43x
2	1024*1024	1.42x	12.49x
3	2048*2048	1.57x	9.12x
4	4000*4000	1.42x	11.46x

GRAPH:

Execution time vs Image Size

Speedup vs Image Size



6.CONCLUSION

CUDA GPU acceleration demonstrates superior performance across all tested image processing techniques—Otsu thresholding, Gaussian filtering, and edge detection (Sobel, Prewitt, and Robert). It consistently outperforms both serial and parallel CPU implementations in execution time, speed-up, and efficiency. For Otsu thresholding, GPU maintains very low and stable execution times, even for large images (e.g., 0.0168s for 4000×4000), with speed-ups reaching up to 12.49x over the serial CPU. Parallel CPU shows modest improvement (up to 2.9x) but struggles to scale efficiently.

In Gaussian filtering, the GPU achieves massive speed-ups—up to 198x—while the parallel CPU reaches only around 4.5x, again confirming the GPU's scalability and efficiency. GPU utilization efficiency also surpasses the CPU, with values near 4.95 compared to the CPU's ~2.2.

For edge detection methods, CUDA delivers exceptional acceleration: up to 38x speed-up for Sobel, 38x for Prewitt on 2048*2048 and 4000*4000 images,whereas CPU has upto 2x which is comparatively highly less than GPU. Even as image sizes grow (e.g., 4000×4000), the GPU retains a clear advantage, maintaining high speed-ups (up to 38x), while CPU-based methods show declining performance gains. Sobel and Prewitt, which involve larger convolution kernels, particularly benefit from GPU parallelism.

Otsu's thresholding demonstrates moderate speedup on parallel CPU and significantly higher speedup on GPU, particularly for larger images.The CUDA GPU version delivers strong performance, achieving up to 12.49x speedup (for 1024×1024) and maintaining 9.12x–11.46x for larger sizes.However, the overall speedup is lower compared to other image processing algorithms, mainly because Otsu's thresholding relies on global histogram computation and iterative variance analysis, which involve fewer parallelizable independent operations,limiting the algorithm's parallel efficiency on both CPU and GPU.

Overall, CUDA GPU is the most effective solution for real-time and high-resolution image processing, offering the best combination of speed, scalability, and resource efficiency, while significantly reducing execution time across all tested algorithms. Parallel CPU serves as a viable middle-ground when GPU is unavailable, delivering moderate improvements, but it cannot match the performance and time efficiency of GPU acceleration.