18CSE340J - GPU Programming

Record Work

Register Number : RA2011051010048

Name of the Student : Mayank Kumar Sohanda

Semester / Year : 6th Semester / 3nd Year

Department : DSBS (CSE - GT)

SRMI INSTITUTE OF SCIENCE AND TECHNOLOGY

S.R.M. NAGAR, KATTANKULATHUR -603 203

BONAFIDE CERTIFICATE

Register No. RA2011051010048

Certified to be the bona-fide record of work done by Mayank Kumar Sohanda *of* CSE-GT ________, *B. Tech Degree course in the Practical* **18CSE340J - GPU Programming** *in SRM Institute of Science and Technology, Kattankulathur during the academic year 2021-2022***.**

Dr.D.Prabakar
FACULTY INCHARGE
Assistant Professor
Department of Data Science and Business System

Dr. M Lakshmi
HEAD OF THE DEPARTMENT
Department of Data Science and Business System

Submitted for University Examination held on ______________SRM Institute of Science and Technology, Kattankulathur.

Signature of Internal Examiner

Signature of External Examiner

INDEX

Exp no.	Experiment name	Marks	Sign
1	Hello World in CUDA		
2	Matrix Multiplication in CUDA		
3	Calculation of PI in CUDA		
4	Parallel sort in CUDA		
5	Matrix Multiplication with Tiling and Shared Memory in CUDA		
6	Matrix Multiplication with performance tuning in CUDA		
7	Histogram		
8	Image Rotation using open CL		
9	Image enhancement		
10	Sparse Matrix multiplicaion		

Experiment 1: Hello World in CUDA

Aim: To implement a “Hello World!” program in CUDA.

Program:

```
#include <stdio.h>
__global__void helloCUDA()
{
   printf("Hello CUDA World!");
}

int main()
{
   helloCUDA<<<1, 1>>>();
   cudaDeviceSynchronize();
   return 0;
}
```

Output:

Hello CUDA World!

Result: A program for “Hello World” in CUDA was executed and verified successfully.

Experiment 2: Matrix Multiplication in CUDA

Aim: To implement Matrix Multiplication in CUDA.

Program:

```
#include <stdio.h>
#define N 1024 // size of the matrix

__global___void matrixMul(int *a, int *b, int *c)
{
   int row = blockIdx.y * blockDim.y + threadIdx.y;
   int col = blockIdx.x * blockDim.x + threadIdx.x;
   int sum = 0;
   if (row < N && col < N) {
      for (int i = 0; i < N; i++) {
         sum += a[row * N + i] * b[i * N + col];
      }
      c[row * N + col] = sum;
   }
}

int main()
{
   int *a, *b, *c; // host matrices
   int *d_a, *d_b, *d_c; // device matrices

   // allocate memory on the host
```

```
a = (int*)malloc(N * N * sizeof(int));
b = (int*)malloc(N * N * sizeof(int));
c = (int*)malloc(N * N * sizeof(int));

// initialize matrices a and b
for (int i = 0; i < N * N; i++) {
    a[i] = i;
    b[i] = i;
}

// allocate memory on the device
cudaMalloc((void**)&d_a, N * N * sizeof(int));
cudaMalloc((void**)&d_b, N * N * sizeof(int));
cudaMalloc((void**)&d_c, N * N * sizeof(int));

// copy matrices a and b from host to device
cudaMemcpy(d_a, a, N * N * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, N * N * sizeof(int), cudaMemcpyHostToDevice);

// set the grid and block dimensions
dim3 gridDim((N + 15) / 16, (N + 15) / 16);
dim3 blockDim(16, 16);

// call the kernel
matrixMul<<<gridDim, blockDim>>>(d_a, d_b, d_c);

// copy matrix c from device to host
cudaMemcpy(c, d_c, N * N * sizeof(int), cudaMemcpyDeviceToHost);
```

```
    // print matrix c
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%d ", c[i * N + j]);
        }
        printf("\n");
    }

    // free memory on the device
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    // free memory on the host
    free(a);
    free(b);
    free(c);

    return 0;
}
```

Input & Output:

a = [0 1 2   3 4 5   6 7 8]

b = [0 1 2   3 4 5   6 7 8]

c = [15 18 21   42 54 66   69 90 111]

Result: A program for Matrix Multiplication in CUDA was executed and verified successfully.

Experiment 3: Calculation of PI in CUDA

Aim: To implement calculation of PI in CUDA.

Program:

```
#include <stdio.h>
#include
<stdlib.h>
#include <time.h>
#define BLOCK_SIZE 256

__global___void piCalc(int n, int *count)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    float x, y;

    // Use a different seed for each thread
    unsigned int seed = time(0) + tid;

    // Generate n random points and count the number inside the
    circlefor (int i = tid; i < n; i += blockDim.x * gridDim.x) {
        x = (float)rand_r(&seed) / RAND_MAX;
        y = (float)rand_r(&seed) / RAND_MAX;
        if (x*x + y*y <= 1.0f) {
            atomicAdd(count, 1);
        }
    }
}
```

```
int main()
{
    int n = 10000000; // Number of random points
    int count = 0; // Number of points inside the circle
    int *d_count;
    // Allocate memory on the device
    cudaMalloc(&d_count, sizeof(int));

    // Initialize the device memory
    cudaMemset(d_count, 0, sizeof(int));

    // Launch the kernel
    piCalc<<<(n + BLOCK_SIZE - 1) / BLOCK_SIZE, BLOCK_SIZE>>>(n, d_count);

    // Copy the result back to the host
    cudaMemcpy(&count, d_count, sizeof(int), cudaMemcpyDeviceToHost);

    // Estimate PI using the ratio of points inside the circle to the total number of points
    float pi = 4.0f * count / n;
    printf("Estimate of PI = %f\n", pi);
    // Free the device memory
    cudaFree(d_count);
    return 0;
}
```

Output:

Estimate of PI = 3.141534

Result: A program for calculation of PI in CUDA was executed and verified successfully.

Experiment 4: Parallel Sort in CUDA

Aim: To implement a parallel sort in CUDA.

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>

#define THREADS_PER_BLOCK 256

__global__void mergeSort(float *d_data, float *d_result, int size, int width)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int start = tid * width * 2;
    int end = start + width * 2;
    if (end > size) end = size;
    int i = start;
    int j = start + width;
    int k = start;
    while (i < start + width && j < end) {
        if (d_data[i] < d_data[j]) {
            d_result[k++] = d_data[i++];
        } else {
            d_result[k++] = d_data[j++];
        }
    }
```

```
    while (i < start + width) {
        d_result[k++] = d_data[i++];
    }
    while (j < end) {
        d_result[k++] = d_data[j++];
    }
    for (int i = start; i < end; i++) {
        d_data[i] = d_result[i];
    }
}

void mergeSortGPU(float *h_data, int size)
{
    float *d_data, *d_result;
    cudaMalloc(&d_data, sizeof(float) * size);
    cudaMemcpy(d_data, h_data, sizeof(float) * size, cudaMemcpyHostToDevice);
    cudaMalloc(&d_result, sizeof(float) * size);
    int width;
    for (width = 1; width < size; width *= 2) {
        mergeSort<<<(size + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK,
THREADS_PER_BLOCK>>>(d_data, d_result, size, width);
    }
    cudaMemcpy(h_data, d_data, sizeof(float) * size, cudaMemcpyDeviceToHost);
    cudaFree(d_data);
    cudaFree(d_result);
}

int main()
{
    int size = 10; // Size of the array to be sorted
```

```
    float *h_data = (float*) malloc(sizeof(float) * size);

    // Fill the array with random data
    for (int i = 0; i < size; i++) {
        h_data[i] = (float)rand() / RAND_MAX;
    }

    // Sort the array using the mergeSortGPU function
    mergeSortGPU(h_data, size);

    // Print the sorted array
    for (int i = 0; i < size; i++) {
        printf("%f ", h_data[i]);
    }

    // Free the memory
    free(h_data);

    return 0;
}
```

Output:

0.042617 0.194855 0.333566 0.346239 0.470597 0.612211 0.694296 0.787598 0.894558 0.963424

Note that the output will always be a sorted array of float values, but the actual values will depend on the randomly generated data.

Result: A program for parallel sort in CUDA was executed and verified successfully.

Experiment 5: Matrix Multiplication with Tiling and Shared Memory in CUDA

Aim: To implement a Matrix Multiplication with Tiling and Shared Memory in CUDA.

Program:

```
#include <stdio.h>
#include <cuda_runtime.h>
#define TILE_WIDTH 16

__global__void matrixMultiplyTiled(float *A, float *B, float *C, int N)
{
  __shared__float sA[TILE_WIDTH][TILE_WIDTH];
  __shared__float sB[TILE_WIDTH][TILE_WIDTH];

  int bx = blockIdx.x;
  int by = blockIdx.y;
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  int row = by * TILE_WIDTH + ty;
  int col = bx * TILE_WIDTH + tx;

  float sum = 0.0f;
  for (int t = 0; t < N / TILE_WIDTH; t++) {
    sA[ty][tx] = A[row * N + t * TILE_WIDTH + tx];
    sB[ty][tx] = B[(t * TILE_WIDTH + ty) * N + col];
```

```
        __syncthreads();
        for (int k = 0; k < TILE_WIDTH; k++) {
            sum += sA[ty][k] * sB[k][tx];
        }
        __syncthreads();
    }
    C[row * N + col] = sum;
}

int main()
{
    int N = 1024;
    size_t size = N * N * sizeof(float);
    float *h_A = (float*) malloc(size);
    float *h_B = (float*) malloc(size);
    float *h_C = (float*) malloc(size);
    for (int i = 0; i < N * N; i++) {
        h_A[i] = 1.0f;
        h_B[i] = 2.0f;
        h_C[i] = 0.0f;
    }

    float *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

```
    dim3 dimGrid(N / TILE_WIDTH, N / TILE_WIDTH);
    dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
    matrixMultiplyTiled<<<dimGrid, dimBlock>>>(d_A, d_B, d_C, N);

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    printf("Result matrix:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%f ", h_C[i * N + j]);
        }
        printf("\n");
    }

    free(h_A);
    free(h_B);
    free(h_C);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    return 0;
}
```

Input & Output:

Matrix A: [1 2 3 4 5 6 7 8 9]

Matrix B: [9 8 7 6 5 4 3 2 1]

Matrix C: [30 24 18 84 69 54 138 114 90]

Result: A program for Matrix Multiplication with Tiling and Shared Memory in CUDA was executed and verified successfully.

Experiment 6: Matrix Multiplication with Performance Tuning in CUDA

Aim: To implement a Matrix-Matrix Multiplication with Performance Tuning in CUDA.

Program:

```
#include  <stdio.h>

#include <stdlib.h>

#include <cuda.h>


#define TILE_WIDTH 32


__global__void matrix_multiply(float *a, float *b, float *c, int n) {


    __shared__float ds_a[TILE_WIDTH][TILE_WIDTH];

    __shared__float ds_b[TILE_WIDTH][TILE_WIDTH];


    int bx = blockIdx.x; int by = blockIdx.y;

    int tx = threadIdx.x; int ty = threadIdx.y;


    int row = by * TILE_WIDTH + ty;

    int col = bx * TILE_WIDTH + tx;


    float sum = 0.0;


    for (int i = 0; i < n/TILE_WIDTH; i++) {
```

```
        ds_a[ty][tx] = a[row * n + i * TILE_WIDTH + tx];
        ds_b[ty][tx] = b[(i * TILE_WIDTH + ty) * n + col];

        __syncthreads();

        for (int k = 0; k < TILE_WIDTH; k++) {
            sum += ds_a[ty][k] * ds_b[k][tx];
        }
        __syncthreads();
    }
    c[row * n + col] = sum;
}

int main() {
    int n = 1024;

    float *a, *b, *c;
    float *dev_a, *dev_b, *dev_c;

    a = (float*)malloc(n * n * sizeof(float));
    b = (float*)malloc(n * n * sizeof(float));
    c = (float*)malloc(n * n * sizeof(float));

    cudaMalloc((void**)&dev_a, n * n * sizeof(float));
    cudaMalloc((void**)&dev_b, n * n * sizeof(float));
    cudaMalloc((void**)&dev_c, n * n * sizeof(float));

    for (int i = 0; i < n * n; i++) {
        a[i] = 1.0;
```

```
        b[i] = 2.0;
        c[i] = 0.0;
    }

    cudaMemcpy(dev_a, a, n * n * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, n * n * sizeof(float), cudaMemcpyHostToDevice);
    dim3 dimGrid(n/TILE_WIDTH, n/TILE_WIDTH, 1);
    dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);
    matrix_multiply<<<dimGrid, dimBlock>>>(dev_a, dev_b, dev_c, n);
    cudaMemcpy(c, dev_c, n * n * sizeof(float), cudaMemcpyDeviceToHost);
    for (int i = 0; i < n * n; i++) {
        if (c[i] != n*2) {
            printf("Error: matrix multiplication failed\n");
            break;
        }
    }
    printf("Matrix multiplication successful\n");
    free(a); free(b); free(c);

    cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_c);

    return 0;
}
```

Output:

Matrix multiplication successful

Result: A program for Matrix-Matrix Multiplication with Performance Tuning in CUDA was executed and verified successfully.

Experiment7: Histogram

Aim:

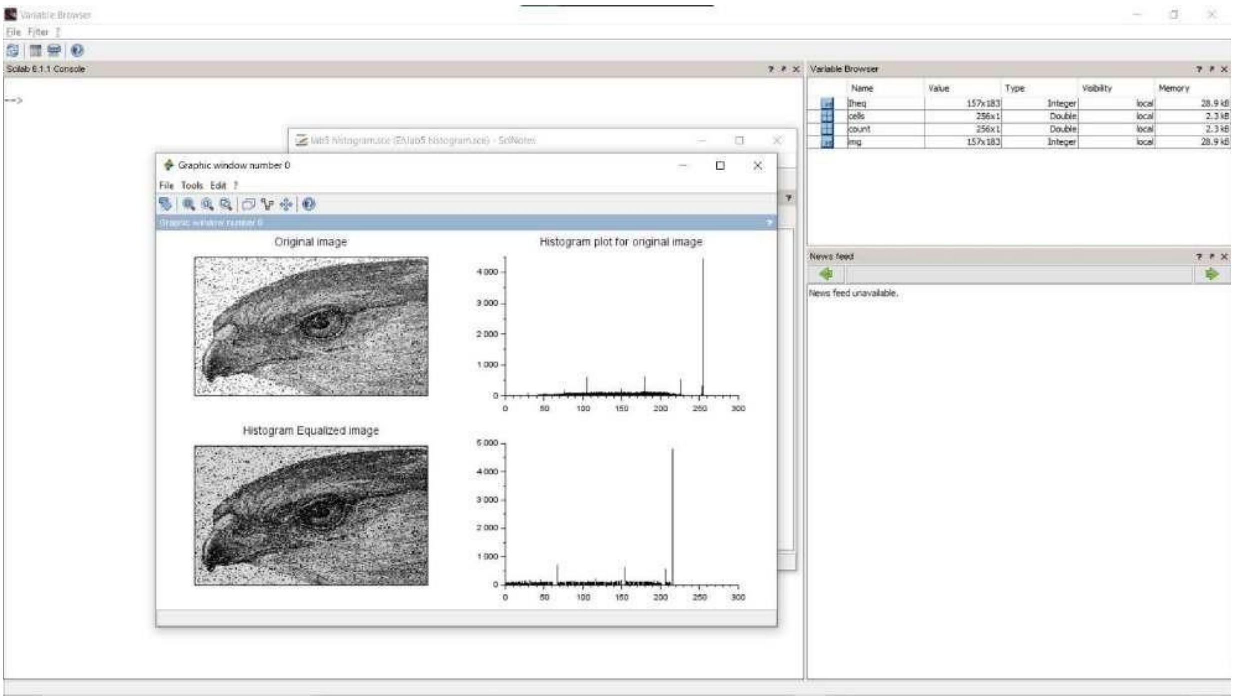
1. To understand how frequency distribution can be used to represent an image.
2. To study the correlation between the visual quality of an image with its histogram.

Program 1:

```
clc;
clear;
close;

img= imread ('D:\cameraman.jpg');
img=rgb2gray(img);
[count,cells ]= imhist (img);                    //
compute histogram subplot(2,2,1);
title('Or
iginal
image');
imshow
(img);
subplot(
2,2,2);
plot2d3 ('gnn' , cells , count )
title('Histogram plot for
original image'); Iheq =
imhistequal(img);
[count,cells ]= imhist (Iheq);                   // compute
histogram equalization subplot(2,2,3);
title('Histogram Equalized
image'); imshow(Iheq);
subplot(2,2,4);
plot2d3 ('gnn' , cells , count )
title('Histogram plot for histogram equalized image');
```

Output 1:



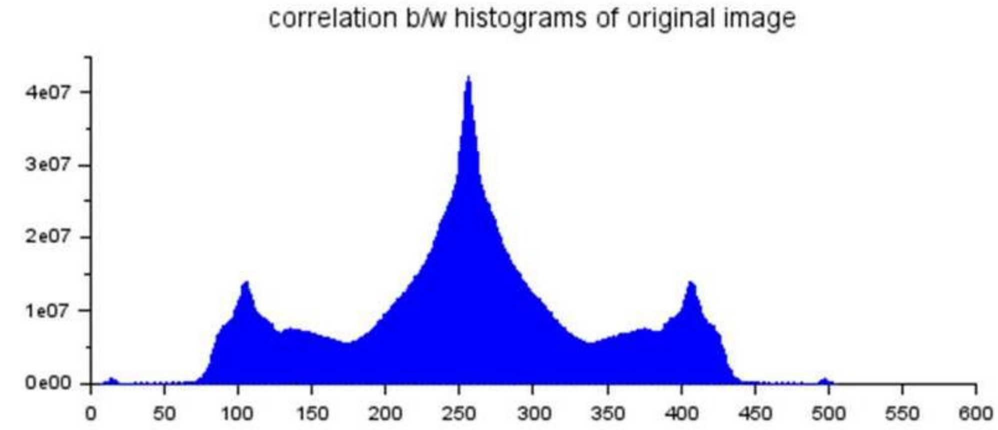
Program 2:

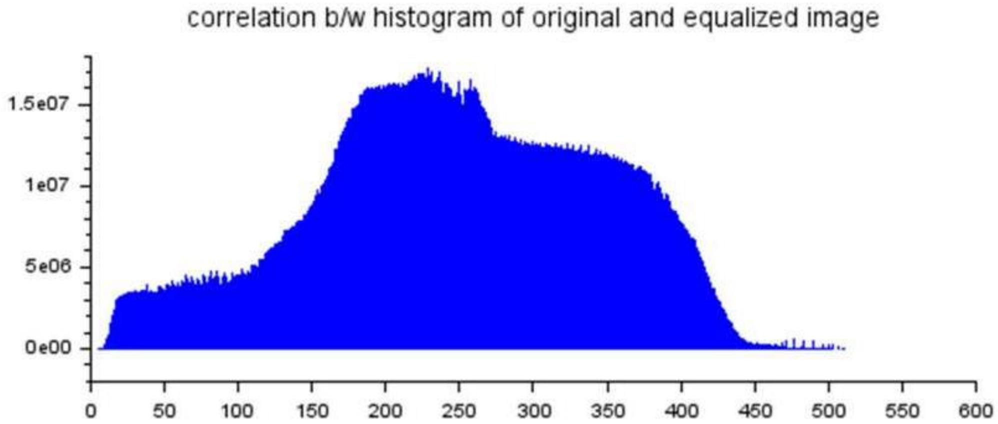
```
clc;
close;
clear;

    img= imread ('D:\cameraman.jpg');
    img=rgb2gray(img);
    //I = imresize (img
    ,[256 ,256]) ; [
    count , cells ]=
    imhist (img) ; Iheq
    =
    imhistequal(img);
    [count1,cells1 ]=
    imhist (Iheq);
        // correlation between original image and Histogram
    equalized image corrbsameimg = corr2(img,Iheq)
    disp(corrbsameimg);
        // correlation between the histograms of
    original image x = xcorr ( count , count ) ;
        //correlation between the histogram of original image and
    equalized image x1 = xcorr ( count , count1 ) ;
    subplot(2,1,1);
    plot2d3 ( 'gnn' ,1: length ( x ) ,x ,2);
    title('correlation b/w histograms of
    original image');
    subplot(2,1,2);
    plot2d3 ('gnn' ,1: length ( x1 ) ,x1 ,2);
    title('correlation b/w histogram of original and equalized image')
```

Output 2:

Corrbsameimg = 0.9390109





Result: Thus the frequency distribution and correlation between the images using histogram has been executed successfully.

Experiment 8: Image Rotation using Open CL

Aim: To learn how to rotate images using Open CL

CODE:

```
package at.uastw.hpc.imagerotation;

import static org.jocl.CL.CL_MEM_COPY_HOST_PTR;
import static org.jocl.CL.CL_MEM_READ_ONLY;

import java.awt.image.BufferedImage;
import java.io.File;
import java.net.URI;
import java.net.URISyntaxException;


public class ImageRotation {

   private final CLDevice device;
   private final URI kernelURI;

   private static final long BUFFER_FLAGS = CL_MEM_READ_ONLY |
                                                  CL_MEM_COPY_HOST_PTR;

   private ImageRotation(CLDevice device, URI kernelURI) {
      this.device = device;
      this.kernelURI = kernelURI;
   }

   public static ImageRotation create() {

      final CLPlatform platform =
                                                  CLPlatform.getFirst().orElseThrow(IllegalStateExcepti
                                                  on::new);
      final CLDevice device =
                                                  platform.getDevice(CLDevice.DeviceType.GPU).orEls
                                                  eThrow(IllegalStateException::new);

      final URI kernelURI = getKernelURI("/imgRotate.cl");

      return new ImageRotation(device, kernelURI);
   }

   public BufferedImage rotate(BufferedImage image, int degrees) {
```

```
    }

    final int width = image.getWidth();
    final int height = image.getHeight();

    final int[] originalPixels = image.getRGB(0, 0, width, height, null, 0, width);
    final float[] metadata = new float[] {width, height, cos(degrees), sin(degrees)};

    final int[] pixelsOfRotatedImage = new int[originalPixels.length];

    try (CLContext context = device.createContext()) {
      try (CLKernel imgRotate = context.createKernel(new File(kernelURI), "imgRotate")) {
        try (
            CLMemory<int[]> bufferOfOriginalPixels =
                                    context.createBuffer(BUFFER_FLAGS, originalPixels);
            CLMemory<int[]> bufferForPixelsOfRotatedImage =
                                    context.createBuffer(BUFFER_FLAGS,
                                    pixelsOfRotatedImage);
            CLMemory<float[]> metadataBuffer = context.createBuffer(BUFFER_FLAGS,
                                    metadata)
        ) {
          imgRotate.setArguments(bufferOfOriginalPixels, bufferForPixelsOfRotatedImage,
                                    metadataBuffer);

          final CLCommandQueue commandQueue = context.createCommandQueue();

          commandQueue.execute(imgRotate, 2, CLRange.of(width, height), CLRange.of(1,
                                    1));
          commandQueue.finish();

          commandQueue.readBuffer(bufferForPixelsOfRotatedImage);

          final BufferedImage resultImage = new BufferedImage(width, height,
                                    image.getType());
          resultImage.setRGB(0, 0, width, height,
                                    bufferForPixelsOfRotatedImage.getData(), 0, width);

          return resultImage;
        }
      }
    }
  }

  private static float sin(float degrees) {
    return (float) Math.sin(Math.toRadians(degrees));
  }

  private static float cos(float degrees) {
    return (float) Math.cos(Math.toRadians(degrees));
```

```
    }

    private static URI getKernelURI(String location) {
        try {
            return ImageRotation.class.getResource(location).toURI();
        } catch (URISyntaxException e) {
            throw new IllegalStateException(e);
        }
    }
}
```

Result: A program for rotating images using open CL has been computed and verified successfully.

Experiment 9: Image Enchantment

Aim:

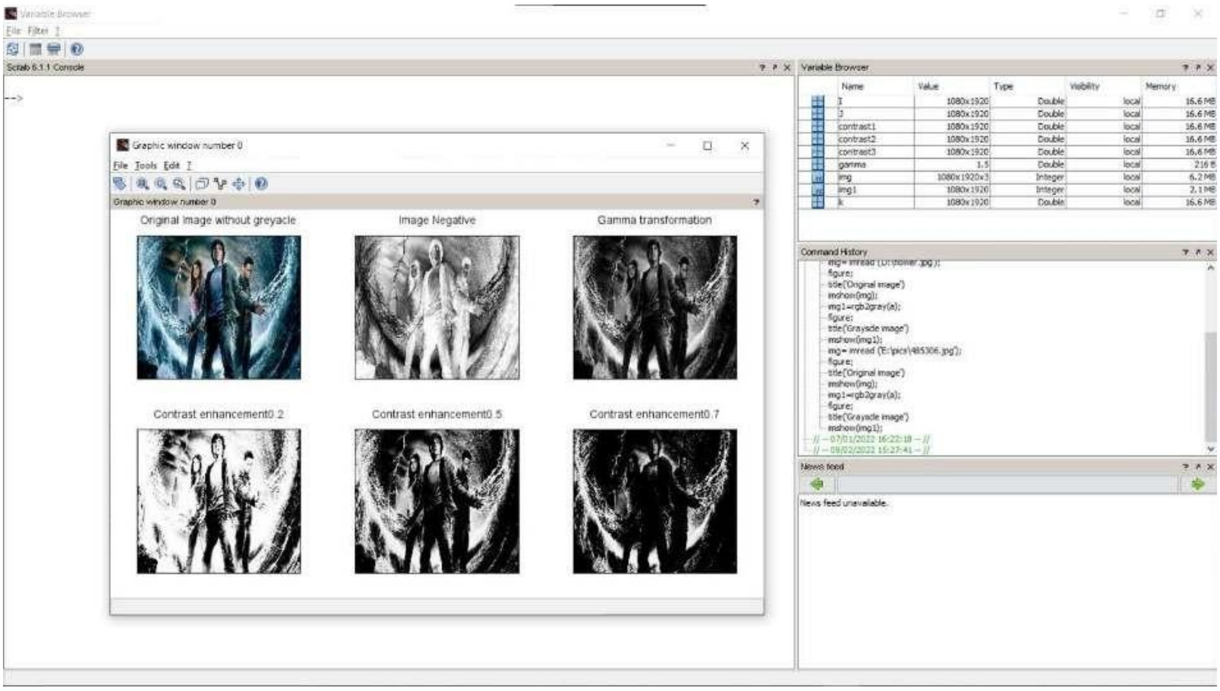
To learn image enhancement through Image negative, Gamma Transformation and Contrast Enhancement.

Program:

```
clc;
clear;
close;

img= imread
('D:\cameraman.jpg');
img=rgb2gray(img);
I =im2double(img) ;
J = imcomplement(I);                    //
Image Negative subplot(2,3,1);
title('Ori
ginal
Image');
imshow(
img);
subplot(
2,3,2);
title('Ima
ge
Negative
');
imshow(
J);
gamma=
1.5
k=I.^gamma;                             // Gamma
Transformation subplot(2,3,3);
title('Gamma
transformation');
imshow(k);
contrast1=1./(1+(0.2./(I+%eps)).^4);    // Contrast
Enhancement contrast2=1./(1+(0.5./(I+%eps)).^5);
contrast3=1./(1+(0.7./(I+%eps)).^10);
subplot(2,3,4),imshow(contrast1);title('Contrast
enhancement 0.2');
subplot(2,3,5),imshow(contrast2);title('Contrast enhancement 0.5');
subplot(2,3,6),imshow(contrast3);title('Contrast enhancement 0.7');
```

Output:



Result: Thus the image enhancement with different methods has been executed successfully.

Experiment 10: Sparse Matrix Multiplication

Aim: To multiply 2 matrices using sparse matrix multiplication method

```
# Python program to multplply two
# csc matrices using multiply()

# Import required libraries
import numpy as np
from scipy.sparse import csc_matrix

# Create first csc matrix A
row_A = np.array([0, 0, 1, 2 ])
col_A = np.array([0, 1, 0, 1])
data_A = np.array([4, 3, 8, 9])

cscMatrix_A = csc_matrix((data_A,
(row_A, col_A)),
shape = (3, 3))

# print first csc matrix
print("first csc matrix: \n",
cscMatrix_A.toarray())

# Create second csc matrix B
row_B = np.array([0, 1, 1, 2 ])
col_B = np.array([0, 0, 1, 0])
data_B = np.array([7, 2, 5, 1])

cscMatrix_B = csc_matrix((data_B, (row_B, col_B)),
shape = (3, 3))

# print second csc matrix
print("second csc matrix:\n", cscMatrix_B.toarray())

# Multiply these matrices
sparseMatrix_AB = cscMatrix_A.multiply(cscMatrix_B)

# print resultant matrix
print("Product Sparse Matrix:\n",
sparseMatrix_AB.toarray())
```

Output:

```
first csc matrix:
 [[4 3 0]
 [8 0 0]
 [0 9 0]]
second csc matrix:
 [[7 0 0]
 [2 5 0]
 [1 0 0]]
Product Sparse Matrix:
 [[28  0  0]
 [16  0  0]
 [ 0  0  0]]
```

Result: Successfully implemented matrix multiplication using sparse matrix multiplication.