600086 Lab Book

Week 4 – CUDA Lab 4. CUDA OpenGL Interoperability & Image processing

Date: 24th Feb 2022

Exercise 1. Create an OpenGL-CUDA program based on a CUDA SDK sample

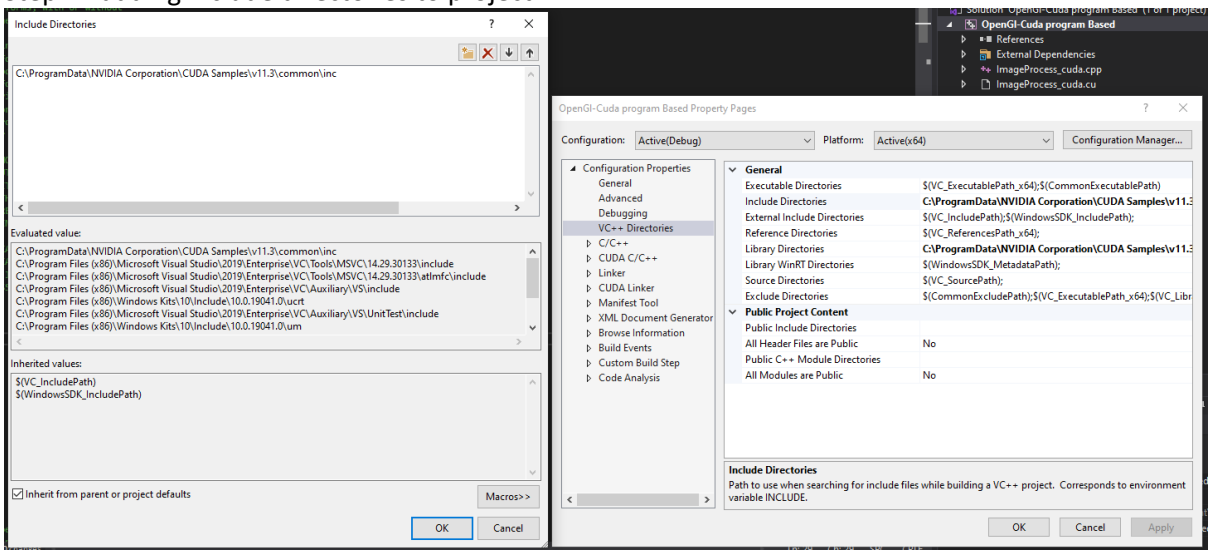
Question:

Create an OpenGL-CUDA program based on a CUDA SDK sample

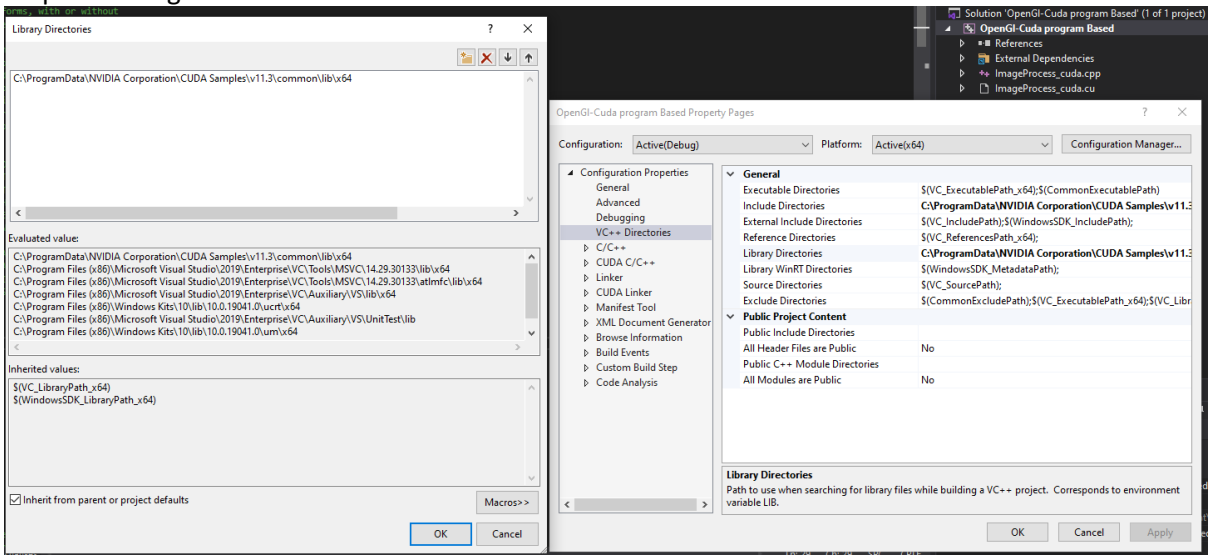
Solution:

No sample code to show

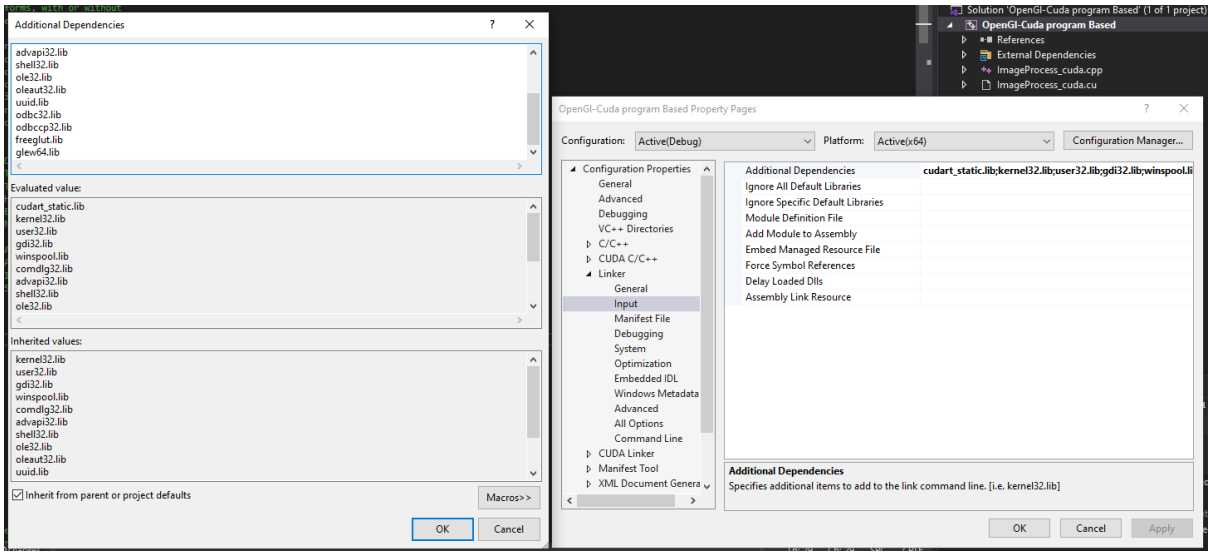
Step 1 : adding include directories to project



Step 2 : adding Lib directories



Step 3 : adding to the Linker files



Step 4 : I then compiled the project resulting in the command line output shown in sample output data

Test data:

n/a

Sample output:

```
Starting Original Texture
GPU Device 0: "Ampere" with compute capability 8.6

CUDA device [NVIDIA GeForce RTX 3070] has 46 Multi-Processors
sdkFindFilePath <lena_bw.pgm> in ./
sdkFindFilePath <lena_bw.pgm> in ./OpenGl-Cuda program Based_data_files/
sdkFindFilePath <lena_bw.pgm> in ./common/
sdkFindFilePath <lena_bw.pgm> in ./common/data/
sdkFindFilePath <lena_bw.pgm> in ./data/
sdkFindFilePath <lena_bw.pgm> in ./src/
sdkFindFilePath <lena_bw.pgm> in ./src/OpenGl-Cuda program Based/data/
sdkFindFilePath <lena_bw.pgm> in ./inc/
sdkFindFilePath <lena_bw.pgm> in ./0_Simple/
sdkFindFilePath <lena_bw.pgm> in ./1_Utilities/
sdkFindFilePath <lena_bw.pgm> in ./2_Graphics/
sdkFindFilePath <lena_bw.pgm> in ./3_Imaging/
sdkFindFilePath <lena_bw.pgm> in ./4_Finance/
sdkFindFilePath <lena_bw.pgm> in ./5_Simulations/
sdkFindFilePath <lena_bw.pgm> in ./6_Advanced/
sdkFindFilePath <lena_bw.pgm> in ./7_CUDALibraries/
sdkFindFilePath <lena_bw.pgm> in ./8_Android/
sdkFindFilePath <lena_bw.pgm> in ./samples/
sdkFindFilePath <lena_bw.pgm> in ./0_Simple/OpenGl-Cuda program Based/data/
sdkFindFilePath <lena_bw.pgm> in ./1_Utilities/OpenGl-Cuda program Based/data/
sdkFindFilePath <lena_bw.pgm> in ./2_Graphics/OpenGl-Cuda program Based/data/
sdkFindFilePath <lena_bw.pgm> in ./3_Imaging/OpenGl-Cuda program Based/data/
sdkFindFilePath <lena_bw.pgm> in ./4_Finance/OpenGl-Cuda program Based/data/
sdkFindFilePath <lena_bw.pgm> in ./5_Simulations/OpenGl-Cuda program Based/data/
sdkFindFilePath <lena_bw.pgm> in ./6_Advanced/OpenGl-Cuda program Based/data/
sdkFindFilePath <lena_bw.pgm> in ./7_CUDALibraries/OpenGl-Cuda program Based/
sdkFindFilePath <lena_bw.pgm> in ./7_CUDALibraries/OpenGl-Cuda program Based/data/
sdkFindFilePath <lena_bw.pgm> in ../
Loaded 'lena_bw.pgm', 512 x 512 pixels

        Controls
        =/- : Zoom in/out
        [esc] - Quit
```

Reflection:

Nothing to report was fairly perfunctory

Metadata:

N/A

Further information:

N/A

Exercise 2. Understand pixel colour

Question:

a) An image is simply a 2D array of pixels. Each pixel has a colour value which can be digitally represented as a list of numbers, depending on the data format adopted. In the framework, the Colour of each pixel is represented in RGBA format using 4 integers, each of which ranging from 0 to 255. Open ImageProcess_cuda.cu and go to the method d_render( ), modify the 4 numbers shown in make_uchar4(..., ..., ..., ...) in the following line:

```
d_output[i] = make_uchar4(c * 0xff, c * 0xff, c * 0xff, 0);
```

say,

```
d_output[i] = make_uchar4(0xff, 0, 0, 0);
```

and then

```
d_output[i] = make_uchar4(0, 0xff, 0, 0);
d_output[i] = make_uchar4(0, 0, 0xff, 0);
```

b) The original image is a grey value image, the pixel intensity at a pixel position at (u,v) is read using float c = tex2DFastBicubic(texObj, u, v); where c is in [0, 1].

c) Now modify the value d_output[i] using image pixel value c read from image location at (u, v) with the following colour and observe how the image colour is changed.

d_output[i] = make_uchar4(0, 0, c*0xff, 0);

Solution:

```
d_output[i] = make_uchar4(c * 0xff, 0, 0, 0);
d_output[i] = make_uchar4(c * 0xff, 0, 0, 0);
d_output[i] = make_uchar4(c * 0xff, 0, 0, 0);
d_output[i] = make_uchar4(c * 0xff, 0, 0, 0);
```

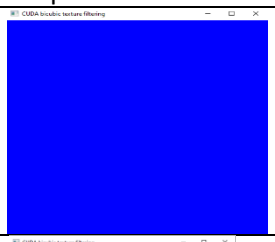
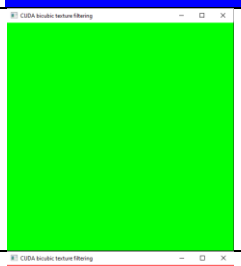
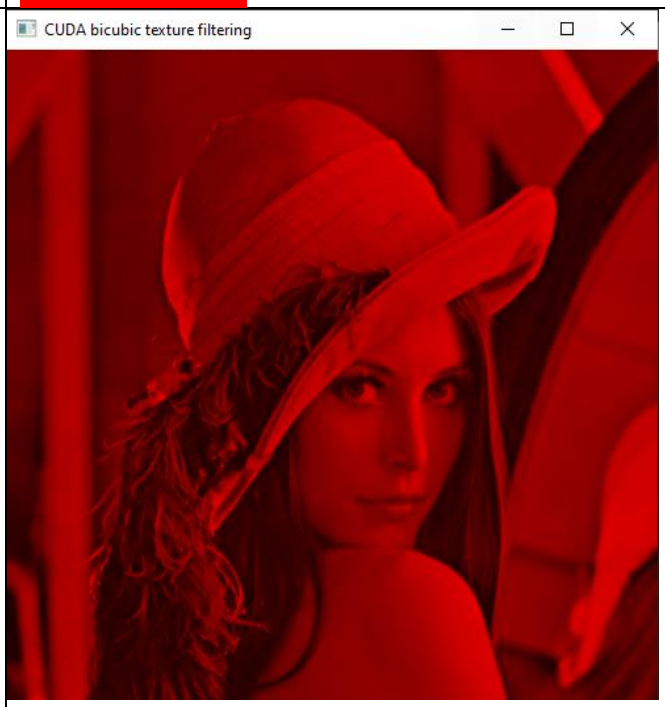
Running each of these one at a time and commenting out the other to display the different resulting outcomes

```
__global__ void d_render(uchar4* d_output, uint width, uint height, float tx,
    float ty, float scale, float cx, float cy,
    cudaTextureObject_t texObj) {
    uint x = blockIdx.x * blockDim.x + threadIdx.x;
    uint y = blockIdx.y * blockDim.y + threadIdx.y;
    uint i = y * width + x;

    float u = (x - cx) * scale + cx + tx;
    float v = (y - cy) * scale + cy + ty;

    if ((x < width) && (y < height)) {
        // write output color
        float c = tex2D<float>(texObj, u, v);

        d_output[i] = make_uchar4(0, 0, c * 0xff, 0);
    }
}
```

Test data:

n/a

Sample output:

code	output
d_output[i] = make_uchar4(0xff, 0, 0, 0);	
d_output[i] = make_uchar4(0, 0xff, 0, 0);	
d_output[i] = make_uchar4(0, 0, 0xff, 0);	
d_output[i] = make_uchar4(0, 0, 0xff, 0);	

Reflection:

Nothing to report was fairly perfunctory

Exercise 3. Image Transformation

Question:

Demonstrate Image transformation.

Solution:

Translate the image.

a. Define a translation as a 2D vector, say float2 T={20, 10};

```
__global__ void d_render(uchar4* d_output, uint width, uint height, float tx,
    float ty, float scale, float cx, float cy,
    cudaTextureObject_t texObj) {
    uint x = blockIdx.x * blockDim.x + threadIdx.x;
    uint y = blockIdx.y * blockDim.y + threadIdx.y;
    uint i = y * width + x;
    float2 T = { 20, 10 };

    if ((x < width) && (y < height)) {
        // write output color

        float c = tex2D<float>(texObj, x, y);
        d_output[i] = make_uchar4(0, 0, c * 0xff, 0);
    }
}
```

b. Translate (x, y) with vector T: x +=T.x; y +=T.y;

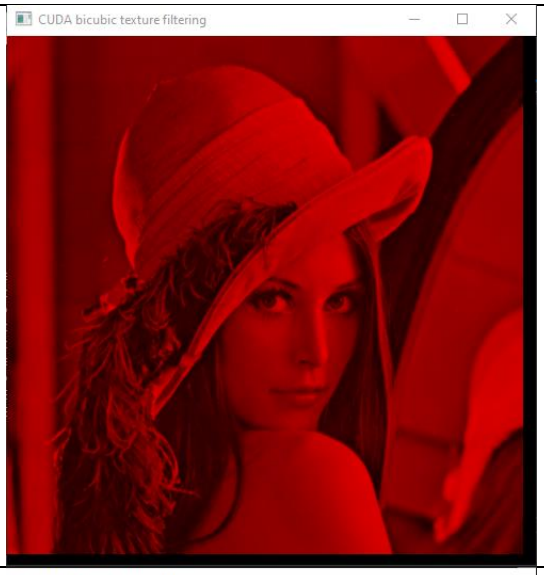
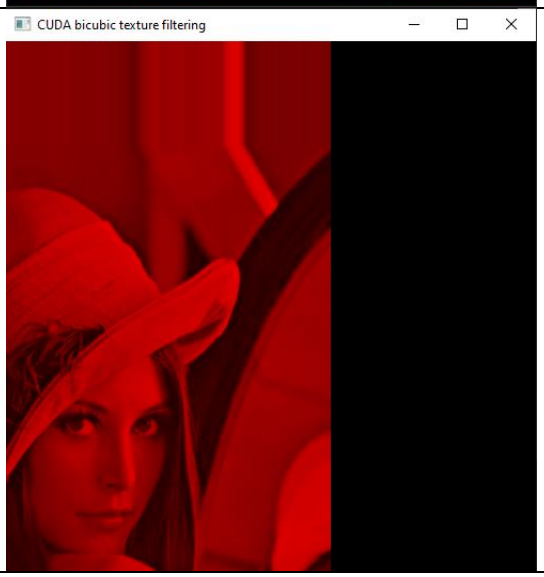
```
__global__ void d_render(uchar4* d_output, uint width, uint height, float tx,
    float ty, float scale, float cx, float cy,
    cudaTextureObject_t texObj) {
    uint x = blockIdx.x * blockDim.x + threadIdx.x;
    uint y = blockIdx.y * blockDim.y + threadIdx.y;
    uint i = y * width + x;
    float2 T = { 20, 10 };
        T:
            x += T.x;
            y += T.y;
            ty;

    if ((x < width) && (y < height)) {
        // write output color

        float c = tex2D<float>(texObj, x, y);
        d_output[i] = make_uchar4(0, 0, c * 0xff, 0);
    }
}
```

c. Read pixel colour with translated coordinates x, y: float c = tex2D(texObj, x, y);

```
__global__ void d_render(uchar4* d_output, uint width, uint height, float tx,
    float ty, float scale, float cx, float cy,
    cudaTextureObject_t texObj) {
    uint x = blockIdx.x * blockDim.x + threadIdx.x;
    uint y = blockIdx.y * blockDim.y + threadIdx.y;
    uint i = y * width + x;
    float2 T = { 20, 10 };
        T:
            x += T.x;
            y += T.y;
            ty;

    if ((x < width) && (y < height)) {
        // write output color

        float c = tex2D<float>(texObj, x, y);
        d_output[i] = make_uchar4(0, 0, c * 0xff, 0);
    }
}
```

d. Compile the run your program and observe if the image is translated according to your wish.
e. Observe how the image is transformed by defining different translation vectors.

Sample output:

Translation	result
`float2 T = { 20, 10 };`	
`float2 T = { 200, -100 };`	

Reflection:

The image has been translated by moving the image in the way described in the vector T. the first value adjusts the translation in the xs axis and the second value adjusts it in the Y axis. When translating to the right the image is replaced by black plixels where the image has moved but when translating to the left the image appears stretched.

Further information:

Why doe the image appear stretched whgen translating in negative directions?

Question:

Demonstrate Image Scaling

Solution:

Scale the image

a. Define a scaling transformation as a 2D vector, say float2 S= {1.2, 0.5};

```
__global__ void d_render(uchar4* d_output, uint width, uint height, float tx,
    float ty, float scale, float cx, float cy,
    cudaTextureObject_t texObj) {
    uint x = blockIdx.x * blockDim.x + threadIdx.x;
    uint y = blockIdx.y * blockDim.y + threadIdx.y;
    uint i = y * width + x;
    float2 T = { 200, -100 };
    float2 S = { 1.2, 0.5 };
        T:
            x += T.x;
            y += T.y;
            ty;

    if ((x < width) && (y < height)) {
        // write output color

        float c = tex2D<float>(texObj, x, y);
        d_output[i] = make_uchar4(0, 0, c * 0xff, 0);
    }
}
```

b. Scale (x, y) with vector S: x *=S.x; y *=S.y;

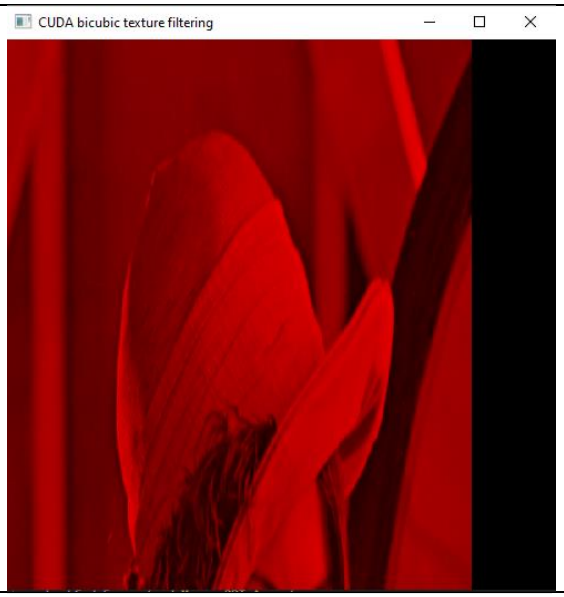
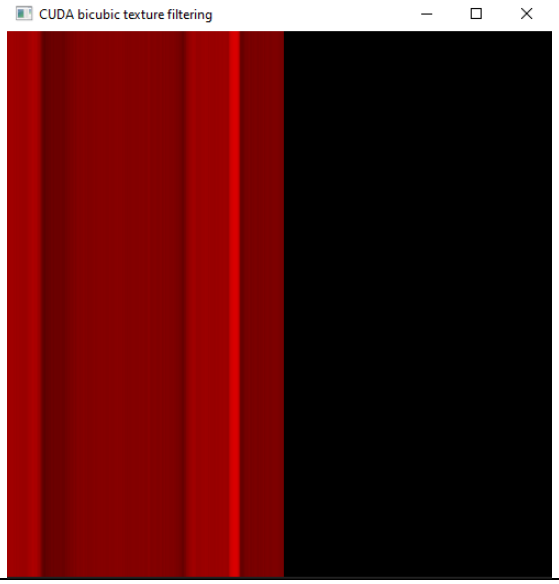
```
__global__ void d_render(uchar4* d_output, uint width, uint height, float tx,
    float ty, float scale, float cx, float cy,
    cudaTextureObject_t texObj) {
    uint x = blockIdx.x * blockDim.x + threadIdx.x;
    uint y = blockIdx.y * blockDim.y + threadIdx.y;
    uint i = y * width + x;
    float2 T = { 200, -100 };
    float2 S = { 1.2, 0.5 };
        T:
            x += T.x;
            y += T.y;
        S:
            x *= S.x;
            y *= S.y;

    if ((x < width) && (y < height)) {
        // write output color

        float c = tex2D<float>(texObj, x, y);
        d_output[i] = make_uchar4(0, 0, c * 0xff, 0);
    }
}
```

c. Read pixel colour with scaled coordinates x, y: float c = tex2D(texObj, x, y);

```
__global__ void d_render(uchar4* d_output, uint width, uint height, float tx,
    float ty, float scale, float cx, float cy,
    cudaTextureObject_t texObj) {
    uint x = blockIdx.x * blockDim.x + threadIdx.x;
    uint y = blockIdx.y * blockDim.y + threadIdx.y;
    uint i = y * width + x;
    float2 T = { 200, -100 };
    float2 S = { 1.2, 0.5 };
        T:
            x += T.x;
            y += T.y;
        S:
            x *= S.x;
            y *= S.y;

    if ((x < width) && (y < height)) {
        // write output color

        float c = tex2D<float>(texObj, x, y);
        d_output[i] = make_uchar4(0, 0, c * 0xff, 0);
    }
}
```

d. Compile the run your program and observe if the image is scaled according to your wish.
e. Observe how the image is scaled by defining different scaling vectors.

Sample output:

Translation	result
`float2 S = { 1.2, 0.5 };`	
`float2 S = { 2, -0.5};`	

Reflection:

The image has been translated by scalin it according to the vector S in the second image I experimented by using a negative value this resulted in a strange image

Further information:

Why does the image appear as shown when scaled in negative directions?

Question:

Demonstrate Image Rotation

Solution:

Rotate the image

a. Define a rotation matrix for a certain rotation angle, float angle = 0.5;

```
__global__ void d_render(uchar4* d_output, uint width, uint height, float tx,
    float ty, float scale, float cx, float cy,
    cudaTextureObject_t texObj) {
    uint x = blockIdx.x * blockDim.x + threadIdx.x;
    uint y = blockIdx.y * blockDim.y + threadIdx.y;
    uint i = y * width + x;
    float2 T = { 0, -0};
    float2 S = { 2, -0.5 };
    float angle = 0.5;
        T:
            x += T.x;
            y += T.y;
        S:
            x *= S.x;
            y *= S.y;

    if ((x < width) && (y < height)) {
        // write output color

        float c = tex2D<float>(texObj, x, y);
        d_output[i] = make_uchar4(0, 0, c * 0xff, 0);
    }
}
```

b. Rotate (x, y) with rotation matrix defined below:

$$R = \begin{pmatrix} \cos(angle) & -\sin(angle) \\ \sin(angle) & \cos(angle) \end{pmatrix}$$

$$float\ rx = x * cos(angle) - y * sin(angle);$$

$$float\ ry = x * sin(angle) + y * cos(angle);$$

```
__global__ void d_render(uchar4* d_output, uint width, uint height, float tx,
    float ty, float scale, float cx, float cy,
    cudaTextureObject_t texObj) {
    uint x = blockIdx.x * blockDim.x + threadIdx.x;
    uint y = blockIdx.y * blockDim.y + threadIdx.y;
    uint i = y * width + x;
    float2 T = { 0, -0};
    float2 S = { 2, -0.5 };
    float angle = 0.5;
        T:
            x += T.x;
            y += T.y;
        S:
            x *= S.x;
            y *= S.y;

    float rx = x * cos(angle) - y * sin(angle);
    float ry = x * sin(angle) + y * cos(angle);

    if ((x < width) && (y < height)) {
        // write output color

        float c = tex2D<float>(texObj, x, y);
        d_output[i] = make_uchar4(0, 0, c * 0xff, 0);
    }
}
```

c. Read pixel colour with scaled coordinates

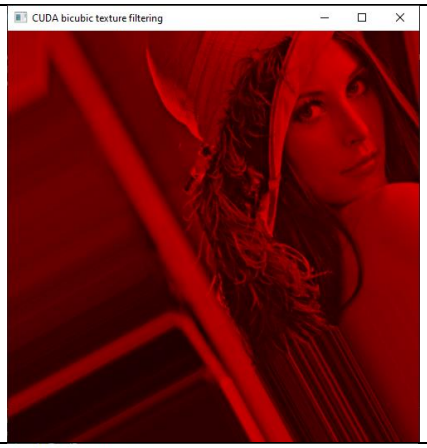
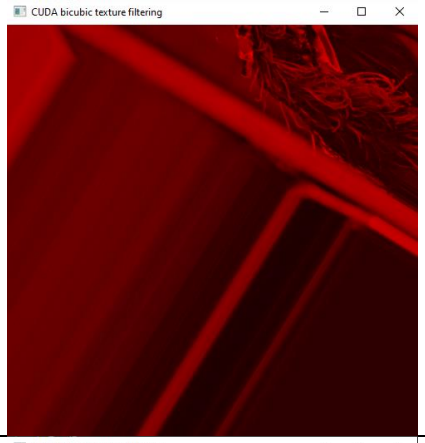
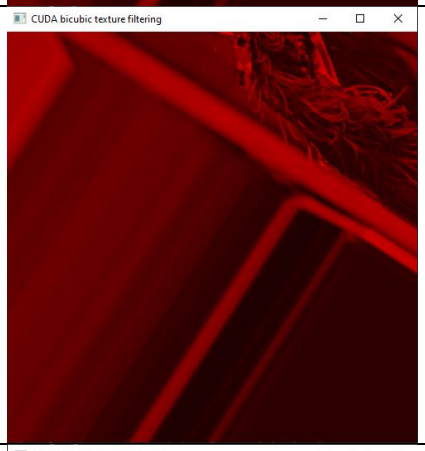
$$uv: float\ c\ =\ tex2D(texObj, rx, ry);$$

```
__global__ void d_render(uchar4* d_output, uint width, uint height, float tx,
    float ty, float scale, float cx, float cy,
    cudaTextureObject_t texObj) {
    uint x = blockIdx.x * blockDim.x + threadIdx.x;
    uint y = blockIdx.y * blockDim.y + threadIdx.y;
    uint i = y * width + x;
    float2 T = { 0, -0};
    float2 S = { 2, -0.5 };
    float angle = 0.5;
        T:
            x += T.x;
            y += T.y;
        S:
            x *= S.x;
            y *= S.y;

    float rx = x * cos(angle) - y * sin(angle);
    float ry = x * sin(angle) + y * cos(angle);

    if ((x < width) && (y < height)) {
        // write output color

        float c = tex2D<float>(texObj, rx, ry);
        d_output[i] = make_uchar4(0, 0, c * 0xff, 0);
    }
}
```

d. Compile the run your program and observe if the image is rotated according to your wish.

e. Further observe how the image is rotated by defining different rotation angles.

Sample output:

Translation	result
`float angle = 0.5;`	
`float angle = 45;`	
`float angle = 1;`	
`float angle = -0.5;`	

Reflection:

The image has been around the origin which is the top left however I am unsure what the angel value equates to it cant be an angle in degrees as 45 results in the same result as 1.

Further information:

What unit does the angle float represent?

Question:

Demonstrate scaling by position

Solution:

Implement the following struct:

$$float\ u\ =\ (x\ -\ cx) * scale\ +\ cx;$$

$$float\ v\ =\ (y\ -\ cy) * scale\ +\ cy;$$

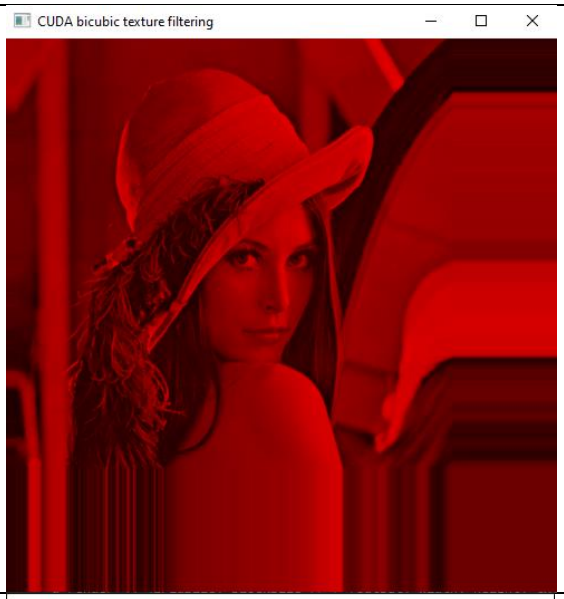
```
__global__ void d_render(uchar4* d_output, uint width, uint height, float tx,
    float ty, float scale, float cx, float cy,
    cudaTextureObject_t texObj) {
    uint x = blockIdx.x * blockDim.x + threadIdx.x;
    uint y = blockIdx.y * blockDim.y + threadIdx.y;
    uint i = y * width + x;
    float2 T = { 0, 0};
    float2 S = { 1, 1 };
    float angle = 0;
        T:
            x += T.x;
            y += T.y;
        S:
            x *= S.x;
            y *= S.y;

    float u = (x - cx) * scale + cx;
    float v = (y - cy) * scale + cy;
    float rx = u * cos(angle) - v * sin(angle);
    float ry = u * sin(angle) + v * cos(angle);



    if ((x < width) && (y < height)) {
        // write output color

        float c = tex2D<float>(texObj, rx, ry);
        d_output[i] = make_uchar4(0, 0, c * 0xff, 0);
    }
}
```

Now the image can be edited by adjusting the values passed into the kernel as shown below:

```
// render image using CUDA
extern "C" void render(int width, int height,  dim3 blockSize, dim3 gridSize,
     uchar4 * output) {

    float tx = 0, ty = 56, scale = 1.3, cx = 35, cy = 0;

        d_render << <gridSize, blockSize >> > (output, width, height, tx, ty, scale,
            cx, cy, rgbaTexdImage);


    getLastCudaError("kernel failed");
}
```

Sample output:

Translation	result

`scale = 1.3,`
`cx = 35,`
`cy = 0;`
	

`scale = 0.5,`
`cx = 35,`
`cy = 1;`
	

Reflection:

Fairly simple

Further information:

None.

Question:

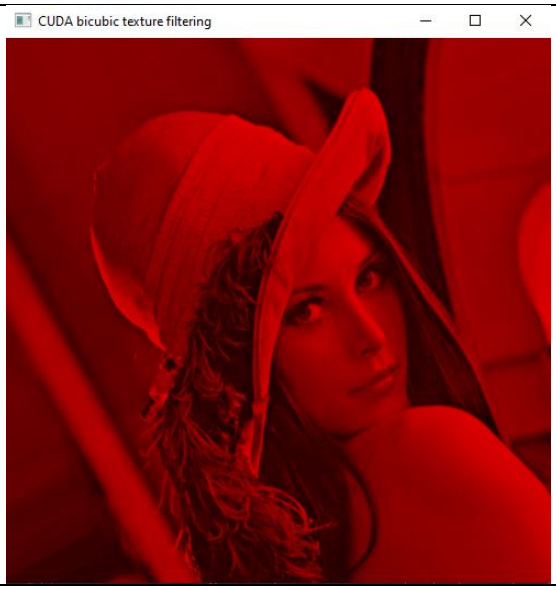
Demonstrate rotation about a centre point

Solution:

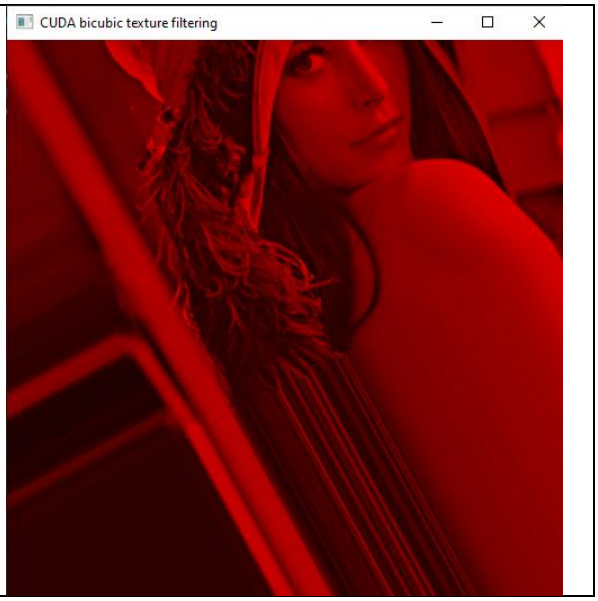
Modify the rotation code to incorporate the passed in centre point values.

```
__global__ void d_render(uchar4* d_output, uint width, uint height, float tx,
    float ty, float scale, float cx, float cy,
    cudaTextureObject_t texObj)
{
    uint x = blockIdx.x * blockDim.x + threadIdx.x;
    uint y = blockIdx.y * blockDim.y + threadIdx.y;
    uint i = y * width + x;

    float angle = 0.5; // angle

    float u = (x - cx) * scale + cx;
    float v = (y - cy) * scale + cy;

    float rx = (x - cx) * cos(angle) - (y - cy) * sin(angle) + cx;
    float ry = (x - cx) * sin(angle) + (y - cy) * cos(angle) + cy;



    if ((x < width) && (y < height)) {
        // write output color

        float c = tex2D<float>(texObj, rx, ry);
        d_output[i] = make_uchar4(0, 0, c * 0xff, 0);
    }
}
```

Sample output:

Translation	result
angle = 0.5, cx = width/2, cy = height/2;	

```
angle = 0.5,
cx = -200,
cy = 150;
```



Reflection:

The rotation calculation had to be adjusted to account for the centre point this is done by performing the calculation on the x value – the cx value and the y – cy value and then the original c values added back to the result as shown in the code sample.

Further information:

None.

Question:

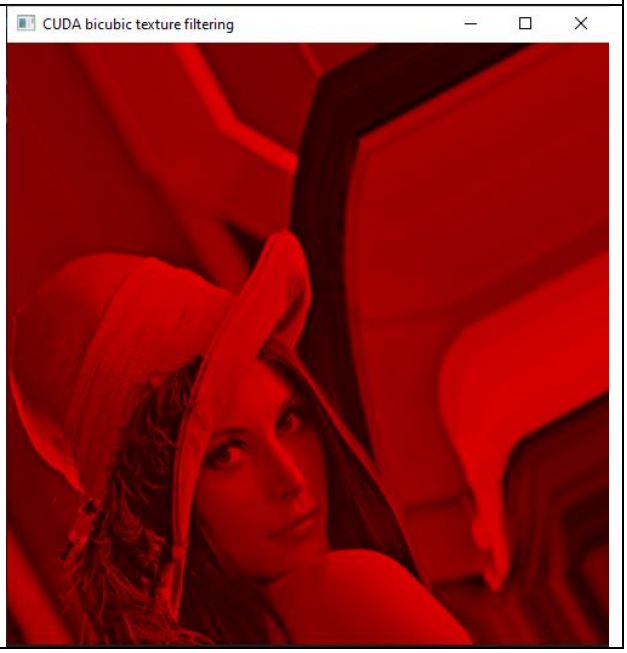
Demonstrate simplified translation implementation.

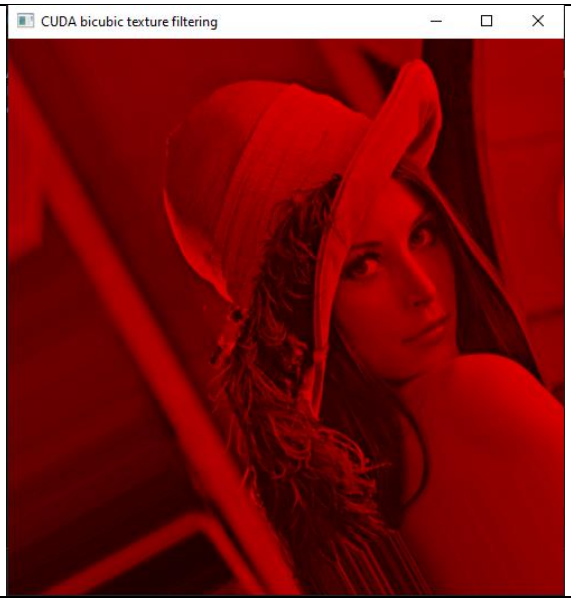
Solution:

The below code implementation will translate the image based on the passed in bvariable to the kernel

```
__global__ void d_render(uchar4* d_output, uint width, uint height, float tx,
    float ty, float scale, float cx, float cy,
    cudaTextureObject_t texObj)
{
    uint x = blockIdx.x * blockDim.x + threadIdx.x;
    uint y = blockIdx.y * blockDim.y + threadIdx.y;
    uint i = y * width + x;

    float angle = 0.5; // angle

    float u = (x - cx) * scale + cx + tx;
    float v = (y - cy) * scale + cy + ty;


    float rx = (u - cx) * cos(angle) - (v - cy) * sin(angle) + cx;
    float ry = (u - cx) * sin(angle) + (v - cy) * cos(angle) + cy;



    if ((x < width) && (y < height)) {
        // write output color

        float c = tex2D<float>(texObj, rx, ry);
        d_output[i] = make_uchar4(0, 0, c * 0xff, 0);
    }
}
```

Sample output:

Translation	result
scale = 1, angle = 0.5, cx = width/2, cy = height/2, tx = 100, ty = -100,	

scale = 1, angle = 0.5, cx = width/2, cy = height/2, tx = -50, ty = 50,	

Reflection:

Fairly perfunctory

Further information:

None.

Exercise 4. Image smoothing

Question:

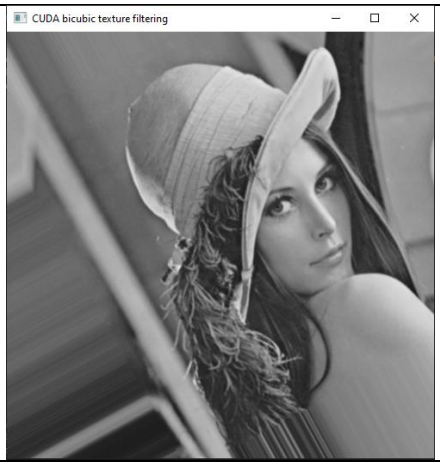
Demonstrate Image smoothing.

Solution:

The below code implements image smoothing using an order 1 square neighbour for reference

```
__global__ void d_render(uchar4* d_output, uint width, uint height, float tx,
    float ty, float scale, float cx, float cy,
    cudaTextureObject_t texObj)
{
    uint x = blockIdx.x * blockDim.x + threadIdx.x;
    uint y = blockIdx.y * blockDim.y + threadIdx.y;

    uint index = y * width + x;

    float angle = 0.5; // angle

    float u = (x - cx) * scale + cx + tx;
    float v = (y - cy) * scale + cy + ty;
    float d = 0.0f;

    float rx = (u - cx) * cos(angle) - (v - cy) * sin(angle) + cx;
    float ry = (u - cx) * sin(angle) + (v - cy) * cos(angle) + cy;

    if ((x < width) && (y < height)) {
        // write output color

        float centre = tex2D< float >(texObj, rx, ry);
        float left = tex2D< float >(texObj, rx - 1, ry);
        float right = tex2D< float >(texObj, rx + 1, ry);
        float up = tex2D< float >(texObj, rx, ry + 1);
        float down = tex2D< float >(texObj, rx, ry - 1);

        d = (centre + left + right + up + down) / 5;
        d_output[index] = make_uchar4(d*0xff, d*0xff, d * 0xff, 0);
    }
}
```

Sample output:

Translation	result
scale = 1, angle = 0.5, cx = width/2, cy = height/2, tx = 100, ty = -100,	

Reflection:

Fairly perfunctory in order to adapt this to smooth by any order then a 2d array of pixels could be used instead of the concrete 5 variable implementation used here. The size of the order of neighbours would need to be passed in to the kernel as an additional variable

Further information:

None.