

600086 Lab Book

Week 2 – Lab B

Date: 17th Feb 2022

Callum Gray

Q1. Understand the block and thread indices

Question:

List the values for the built-in variables `threadIdx.x` and `blockIdx.x` corresponding to the following thread configurations used for executing the kernel `addKernel()` function on GPU:

- 1) `addKernel << <1, 5 >> > (dev_c, dev_a, dev_b);`
- 2) `addKernel << <2, 3 >> > (dev_c, dev_a, dev_b);`
- 3) `addKernel << <3, 2 >> > (dev_c, dev_a, dev_b);`
- 4) `addKernel << <6, 1 >> > (dev_c, dev_a, dev_b);`

Solution:

```
__global__ void addKernel(int *c, const int *a, const int *b)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
    printf("Thread ID : %d Block: ID: %d", threadIdx.x, blockIdx.x);
}
```

In the below image each add kernel function is run separately and the others commented out .

```
addKernel << <1, 5 >> > (dev_c, dev_a, dev_b);
addKernel << <2, 3 >> > (dev_c, dev_a, dev_b);
addKernel << <3, 2 >> > (dev_c, dev_a, dev_b);
addKernel << <6, 1 >> > (dev_c, dev_a, dev_b);

cudaDeviceSynchronize();
cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);

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

printf("{1,2,3,4,5} + {10,20,30,40,50} = {%d,%d,%d,%d,%d}\n", c[0], c[1], c[2], c[3], c[4]);
```

Test data:

```
const int size = 5;
const int a[size] = { 1, 2, 3, 4, 5 };
const int b[size] = { 10, 20, 30, 40, 50 };
```

Sample output:

Thread Config	Output
addKernel <<<1, 5 >>> (dev_c, dev_a, dev_b);	Thread ID : 0 Block: ID: 0 Thread ID : 1 Block: ID: 0 Thread ID : 2 Block: ID: 0 Thread ID : 3 Block: ID: 0 Thread ID : 4 Block: ID: 0
addKernel <<<2, 3 >>> (dev_c, dev_a, dev_b);	Thread ID : 0 Block: ID: 0 Thread ID : 1 Block: ID: 0 Thread ID : 2 Block: ID: 0 Thread ID : 0 Block: ID: 1 Thread ID : 1 Block: ID: 1 Thread ID : 2 Block: ID: 1
addKernel <<<3, 2 >>> (dev_c, dev_a, dev_b);	Thread ID : 0 Block: ID: 0 Thread ID : 1 Block: ID: 0 Thread ID : 0 Block: ID: 2 Thread ID : 1 Block: ID: 2 Thread ID : 0 Block: ID: 1 Thread ID : 1 Block: ID: 1
addKernel <<<6, 1 >>> (dev_c, dev_a, dev_b);	Thread ID : 0 Block: ID: 5 Thread ID : 0 Block: ID: 0 Thread ID : 0 Block: ID: 3 Thread ID : 0 Block: ID: 2 Thread ID : 0 Block: ID: 4 Thread ID : 0 Block: ID: 1

Reflection:

The block number indicates which thread block the thread is located in and the thread number indicates which thread of the block is currently executing. This could be useful to identify threads when in a large-scale process where the parallel processing becomes more complex.

Metadata:

Threadidx.x, blockidx.x

Further information:

N/A

Q2. Exercise 2. Find vector addition using multiple 1D thread blocks

Question:

For the vector addition problem considered in the CUDA template, find the solution based on the following thread configurations by modifying the following line of CUDA code:

```
int i = threadIdx.x;
```

- 1) addKernel <<<2, 3 >>> (dev_c, dev_a, dev_b);
- 2) addKernel <<<3, 2 >>> (dev_c, dev_a, dev_b);
- 3) addKernel <<<6, 1 >>> (dev_c, dev_a, dev_b);

Solution:

```
__global__ void addKernel(int *c, const int *a, const int *b)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
    printf("Thread ID : %d Block: ID: %d", threadIdx.x, blockIdx.x);
}
```

In the below image each add kernel function is run separately and the others commented out .

```
addKernel <<<2, 3 >>> (dev_c, dev_a, dev_b);
addKernel <<<3, 2 >>> (dev_c, dev_a, dev_b);
addKernel <<<6, 1 >>> (dev_c, dev_a, dev_b);

cudaDeviceSynchronize();
cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);

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

printf("{1,2,3,4,5} + {10,20,30,40,50} = {%d,%d,%d,%d,%d}\n", c[0], c[1], c[2], c[3], c[4]);
```

Test data:

```
const int size = 5;
const int a[size] = { 1, 2, 3, 4, 5 };
const int b[size] = { 10, 20, 30, 40, 50 };
```

Sample output:

Thread Config	Output
addKernel <<<2, 3 >>> (dev_c, dev_a, dev_b);	{1,2,3,4,5} + {10,20,30,40,50} = {11,22,33,0,0}
addKernel <<<3, 2 >>> (dev_c, dev_a, dev_b);	{1,2,3,4,5} + {10,20,30,40,50} = {11,22,0,0,0}
addKernel <<<6, 1 >>> (dev_c, dev_a, dev_b);	{1,2,3,4,5} + {10,20,30,40,50} = {11,0,0,0,0}

Reflection:

The output is limited by the number of threads generated in in each block as the calculation uses the thread ID to index each array. SO if more than 5 arrays are used then an error will be raised.

Metadata:

Multithread

Further information:

N/A

Q3. Exercise 3. Understand the thread indices for 2D blocks

Question:

List the values for the built-in variables **threadIdx.x** and **threadIdx.y** corresponding to the following thread configurations used for executing the kernel *addKernel()* function on GPU:

- 1) `addKernel << <1, dim3(2, 3) >> > (dev_c, dev_a, dev_b);`
- 2) `addKernel << <1, dim3(3, 3) >> > (dev_c, dev_a, dev_b);`
- 3) `addKernel << <1, dim3(5, 1) >> > (dev_c, dev_a, dev_b);`

Solution:

```
__global__ void addKernel(int* c, const int* a, const int* b)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
    printf("Thread ID : (X = %d,Y = %d) Block: ID: %d\n", threadIdx.x, threadIdx.y, blockIdx.x);
}
```

In the below image each add kernel function is run separately and the others commented out .

```
addKernel << <1, dim3(2,3) >> > (dev_c, dev_a, dev_b);
addKernel << <1, dim3(3,3) >> > (dev_c, dev_a, dev_b);
addKernel << <1, dim3(5,1) >> > (dev_c, dev_a, dev_b);

cudaDeviceSynchronize();
cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);

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

printf("{1,2,3,4,5} + {10,20,30,40,50} = {%d,%d,%d,%d,%d}\n", c[0], c[1], c[2], c[3], c[4]);
```

Test data:

```
const int size = 5;
const int a[size] = { 1, 2, 3, 4, 5 };
const int b[size] = { 10, 20, 30, 40, 50 };
```

Sample output:

Thread Config	Output
addKernel << <1, dim3(2,3) >> > (dev_c, dev_a, dev_b);	Thread ID : (X = 0,Y = 0) Block: ID: 0 Thread ID : (X = 1,Y = 0) Block: ID: 0 Thread ID : (X = 0,Y = 1) Block: ID: 0 Thread ID : (X = 1,Y = 1) Block: ID: 0 Thread ID : (X = 0,Y = 2) Block: ID: 0 Thread ID : (X = 1,Y = 2) Block: ID: 0
addKernel << <1, dim3(3,3) >> > (dev_c, dev_a, dev_b);	Thread ID : (X = 0,Y = 0) Block: ID: 0 Thread ID : (X = 1,Y = 0) Block: ID: 0 Thread ID : (X = 2,Y = 0) Block: ID: 0 Thread ID : (X = 0,Y = 1) Block: ID: 0 Thread ID : (X = 1,Y = 1) Block: ID: 0 Thread ID : (X = 2,Y = 1) Block: ID: 0 Thread ID : (X = 0,Y = 2) Block: ID: 0 Thread ID : (X = 1,Y = 2) Block: ID: 0 Thread ID : (X = 2,Y = 2) Block: ID: 0
addKernel << <1, dim3(5,1) >> > (dev_c, dev_a, dev_b);	Thread ID : (X = 0,Y = 0) Block: ID: 0 Thread ID : (X = 1,Y = 0) Block: ID: 0 Thread ID : (X = 2,Y = 0) Block: ID: 0 Thread ID : (X = 3,Y = 0) Block: ID: 0 Thread ID : (X = 4,Y = 0) Block: ID: 0

Reflection:

The aim of the above task is to display how to Identify individual threads within a multi-dimensional block. Each thread has a unique Id based on its location in the thread which can be thought of as a coordinate to locate it e.g 2d array of threads has threadIdx.x and threadIdx.y and a 3D array of threads has threadIdx.x ,threadIdx.y and threadIdx.z to locate it.

Metadata:

Multithread threadIdx.x threadIdx.y threadIdx.z

Further information:

N/A

Q4. Exercise 4. Find vector addition using one 2D thread block

Question:

For the vector addition problem considered in the CUDA template, find the solution based on the following thread configurations by modifying the following line of CUDA code:

```
int i = threadIdx.x;
```

- 1) addKernel << <1, dim3(2, 3) >> > (dev_c, dev_a, dev_b);
- 2) addKernel << <1, dim3(3, 2) >> > (dev_c, dev_a, dev_b);
- 3) addKernel << <1, dim3(5, 1) >> > (dev_c, dev_a, dev_b);

Solution:

```
__global__ void addKernel(int* c, const int* a, const int* b)
{
    int i = threadIdx.x + threadIdx.y + (threadIdx.x * (blockDim.y-1));
    c[i] = a[i] + b[i];
    printf("Thread ID : (%d,%d) Block: ID: (%d)\n", threadIdx.x, threadIdx.y, blockIdx.x);
}
```

In the below image each add kernel function is run separately and the others commented out.

```
addKernel << <1, dim3(2,3) >> > (dev_c, dev_a, dev_b);
addKernel << <1, dim3(3,2) >> > (dev_c, dev_a, dev_b);
addKernel << <1, dim3(5,1) >> > (dev_c, dev_a, dev_b);

cudaDeviceSynchronize();
cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);

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

printf("{1,2,3,4,5} + {10,20,30,40,50} = {%d,%d,%d,%d,%d}\n", c[0], c[1], c[2], c[3], c[4]);
```

Test data:

```
const int size = 5;
const int a[size] = { 1, 2, 3, 4, 5 };
const int b[size] = { 10, 20, 30, 40, 50 };
```

Sample output:

Thread Config	Output
addKernel << <1, dim3(2,3) >> > (dev_c, dev_a, dev_b);	Thread ID : (0,0) Block: ID: (0) Thread ID : (1,0) Block: ID: (0) Thread ID : (0,1) Block: ID: (0) Thread ID : (1,1) Block: ID: (0) Thread ID : (0,2) Block: ID: (0) Thread ID : (1,2) Block: ID: (0) {1,2,3,4,5} + {10,20,30,40,50} = {11,22,33,44,55}
addKernel << <1, dim3(3,2) >> > (dev_c, dev_a, dev_b);	Thread ID : (0,0) Block: ID: (0) Thread ID : (1,0) Block: ID: (0) Thread ID : (2,0) Block: ID: (0) Thread ID : (0,1) Block: ID: (0) Thread ID : (1,1) Block: ID: (0) Thread ID : (2,1) Block: ID: (0) {1,2,3,4,5} + {10,20,30,40,50} = {11,22,33,44,55}
addKernel << <1, dim3(5,1) >> > (dev_c, dev_a, dev_b);	Thread ID : (0,0) Block: ID: (0) Thread ID : (1,0) Block: ID: (0) Thread ID : (2,0) Block: ID: (0) Thread ID : (3,0) Block: ID: (0) Thread ID : (4,0) Block: ID: (0) {1,2,3,4,5} + {10,20,30,40,50} = {11,22,33,44,55}

Reflection:

The tasks' goal was to perform vector addition using a multi-dimensional array of threads however the vector addition only occurs on the column number for the thread blocks.

Metadata:

Multithread threadIdx.x threadIdx.y threadIdx.z

Further information:

Unsure how to resolve issue of only adding columns do we need to add an id checking method that will adapt to include rows as an overflow when there aren't enough columns?

Q5. Exercise 5. Find vector addition using multiple 2D thread blocks

Question:

For the vector addition problem considered in the CUDA template, find the solution based on the following thread configurations by modifying the following line of CUDA code:

```
int i = threadIdx.x;
```

- 1) addKernel << < dim3(1, 3), dim3(3, 1) >> > (dev_c, dev_a, dev_b);
- 2) addKernel << < dim3(2, 3), dim3(2, 2) >> > (dev_c, dev_a, dev_b);
- 3) addKernel << < dim3(2, 2), dim3(2, 3) >> > (dev_c, dev_a, dev_b);

Solution:

```
__global__ void addKernel(int* c, const int* a, const int* b)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
    printf("Thread ID : (%d,%d) Block: ID: (%d,%d)\n", threadIdx.x, threadIdx.y, blockIdx.x, blockIdx.y);
}
```

In the below image each add kernel function is run separately and the others commented out.

```
addKernel << <dim3(1,3), dim3(2,3) >> > (dev_c, dev_a, dev_b);
addKernel << <dim3(2,3), dim3(3,2) >> > (dev_c, dev_a, dev_b);
addKernel << <dim3(2,2), dim3(5,1) >> > (dev_c, dev_a, dev_b);

cudaDeviceSynchronize();
cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);

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

printf("{1,2,3,4,5} + {10,20,30,40,50} = {%d,%d,%d,%d,%d}\n", c[0], c[1], c[2], c[3], c[4]);
```

Test data:

```
const int size = 5;
const int a[size] = { 1, 2, 3, 4, 5 };
const int b[size] = { 10, 20, 30, 40, 50 };
```

Sample output:

Thread Config	Output
addKernel << < dim3(1, 3), dim3(3, 1) >> > (dev_c, dev_a, dev_b);	
addKernel << < dim3(2, 3), dim3(2, 2) >> > (dev_c, dev_a, dev_b);	
addKernel << < dim3(2, 2), dim3(2, 3) >> > (dev_c, dev_a, dev_b);	

Reflection:

This is threading 101

Metadata:

Threads

Further information:

Unsure of the use of mut?

600086 Lab Book

Week 3 – Lab C

Date: 22nd Feb 2022

E1. Vector dot-product

Question:

(1). CPU only solution.

Write a C++ program to calculate the dot-product of two vectors used in the template CUDA program created in VS2019.

Solution:

```
int main()
{
    const int arraySize = 5;
    const int a[arraySize] = { 1, 2, 3, 4, 5 };
    const int b[arraySize] = { 10, 20, 30, 40, 50 };
    int c = 0;

    printf("array A = {1,2,3,4,5}\n");
    printf("array B = {10,20,30,40,50}\n");

    for (int i = 0; i < arraySize; i++)
    {
        c += a[i] * b[i];
    }

    printf("the dot product of arrays A and B is: %d", c);
}
```

Test data:

A	1	2	3	4	5
B	10	20	30	40	50

Sample output:

```
array A = {1,2,3,4,5}
array B = {10,20,30,40,50}
the dot product of arrays A and B is: 550
```

Question:

(2). CPU + GPU solution.

The dot-product of vectors $\mathbf{a}=(a_0, a_1, a_2, \dots, a_{n-1})$ and $\mathbf{b}=(b_0, b_1, b_2, \dots, b_{n-1})$, can be found in two steps:

Step 1. Per-element multiplication: In this step, we calculate a vector \mathbf{c} :

$$\mathbf{c} = (a_0b_0, \quad a_1b_1, \quad a_2b_2, \quad \dots, \quad a_{n-1}b_{n-1})$$

This task can be done in parallel on GPU.

Step 2. Calculate on CPU the sum of elements of vector \mathbf{c} found in step 1.

Write a CUDA program to accelerate the calculation of the dot-product by doing the per-element multiplication on the GPU.

Solution:

```
__global__ void addKernel(int *c, int *a, int *b)
{
    int i = threadIdx.x;
    c[i] = a[i] * b[i];
}

const int size = 12;
int a[size] = { 0 };
int b[size] = { 0 };
int c[size] = { 0 };
int d = 0;

for (int i = 0; i < size; i++)
{
    a[i] = i + 1;
    b[i] = (i + 1) * 10;
}

int *dev_a = 0;
int *dev_b = 0;
int *dev_c = 0;

addKernel << 1, size >> > (dev_c, dev_a, dev_b);

cudaDeviceSynchronize();
cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);
cudaFree(dev_c);
cudaFree(dev_a);
cudaFree(dev_b);
printf("Array A = {%d", a[0]);
for (int i = 1; i < size; i++){
    printf(",%d", a[i]);
}
printf("\n");
printf("Array B = {%d", b[0]);
for (int i = 1; i < size; i++){
    printf(",%d", b[i]);
}
printf("\n");
printf("Array C = {%d", c[0]);
for (int i = 1; i < size; i++) {
    printf(",%d", c[i]);
}
printf("\n");
for (int i = 0; i < size; i++){
    d += c[i];
}
printf("the dot product of A & B is: %d", d);
```

Test data:

A	1	2	3	4	5	6	7	8	9	10	11	12
B	10	20	30	40	50	60	70	80	90	100	110	120

Sample output:

```
Array A = {1,2,3,4,5,6,7,8,9,10,11,12}  
Array B = {10,20,30,40,50,60,70,80,90,100,110,120}  
Array C = {10,40,90,160,250,360,490,640,810,1000,1210,1440}  
the dot product of A & B is: 6500
```

Reflection:

This is a very quick technique for calculating the dot product when increasing the lengths of the arrays used in the C++ method the completion time increases greatly whereas with the GPU solution the lengths of the test data array can be increased with almost no difference to the execution times.

Metadata:

"Dot-Product","thread","c++","GPU"

Further information:

E2. Vector dot-product using unified memory

Question:

Exercise 2.1 Vector dot-product using managed memory

Solution:

```
__global__ void PerElement_At看imesB(int *c, int *a, int *b)
{
    c[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];
}

int main()
{
    cudaError_t cudaStatus;
    // Choose which GPU to run on, change this on a multi-GPU system.
    cudaStatus = cudaSetDevice(0);
    const int size = 5;
    int* c, * a, * b;
    cudaMallocManaged(&a, size * sizeof(int));
    cudaMallocManaged(&b, size * sizeof(int));
    cudaMallocManaged(&c, size * sizeof(int));
    for (int i = 0; i < size; i++)
    {
        a[i] = i + 1;
        b[i] = (i + 1) * 10;
        c[i] = 0;
    }
    PerElement_AtimesB << 1, size >> > (c, a, b);
    cudaDeviceSynchronize();
    printf("A is :{ %d", a[0]);
    for(int i = 1; i < size; i++){
        printf(",%d", a[i]);
    }
    printf(" }\n");
    printf("B is :{ %d", b[0]);
    for (int i = 1; i < size; i++){
        printf(",%d", b[i]);
    }
    printf(" }\n");
    printf("C is :{ %d", c[0]);
    for (int i = 1; i < size; i++){
        printf(",%d", c[i]);
    }
    printf(" }\n");
    int d = c[0] + c[1] + c[2] + c[3] + c[4];
    printf("%d", d);
    cudaFree(c);
    cudaFree(a);
    cudaFree(b);
    cudaDeviceReset();
    return 0;
}
```

Test data:

```
A is :{ 1,2,3,4,5 }
B is :{ 10,20,30,40,50 }
```

Sample output:

```
A is :{ 1,2,3,4,5 }
B is :{ 10,20,30,40,50 }
C is :{ 10,40,90,160,250 }
550
```

Question:

Exercise 2.3 Vector dot-product using GPU-declared `__managed__` memory

Solution:

```
__device__ __managed__ int a[5], b[5], c[5];

__global__ void PerElement_At看imesB(int* c, int* a, int* b)
{
    c[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];
}

int main()
{
    cudaSetDevice(0);

    for (int i = 0; i < 5; i++)
    {
        a[i] = i+1;
        b[i] = (i+1) * 10;
    }

    PerElement_AtimesB << 1, 5 >> > (c, a, b);

    cudaDeviceSynchronize();

    printf("A is :{ %d", a[0]);
    for (int i = 1; i < 5; i++) {
        printf(",%d", a[i]);
    }
    printf(" }\n");
    printf("B is :{ %d", b[0]);
    for (int i = 1; i < 5; i++) {
        printf(",%d", b[i]);
    }
    printf(" }\n");
    printf("C is :{ %d", c[0]);
    for (int i = 1; i < 5; i++) {
        printf(",%d", c[i]);
    }
    printf(" }\n");
    int d = c[0] + c[1] + c[2] + c[3] + c[4];
    printf("dot product of A & B is: %d", d);

    cudaDeviceReset();
    return 0;
}
```

Test data:

```
A is :{ 1,2,3,4,5 }
B is :{ 10,20,30,40,50 }
```

Sample output:

```
A is :{ 1,2,3,4,5 }
B is :{ 10,20,30,40,50 }
C is :{ 10,40,90,160,250 }
dot product of A & B is: 550
```

Reflection:

Declaring a shared variable makes the code much neater and easier to read I struggled getting `cudaMallocManaged()` to work as I had expected it however the declared shared memory worked immediately and felt much more intuitive. Added the for loops on the print statements in each section as I had built them with extension in mind so the size characteristic can be adjusted to suit larger vectors and the code will run without fault limited purely by the thread limit.

Metadata:

"Dot-Product","thread","c++","GPU","declared","managed"

Further information:

E3. Vector dot-product using shared Memory

Question:

Task 1. Threads synchronization.

Analyse the above process and identify areas where thread execution needs to be synchronized by calling CUDA function: `__syncthreads()`;

Solution:

```
__device__ __managed__ int a[8], b[8], c[8];

__shared__ int dataPerBlock[4];

__global__ void PerElement_At看imesB(int* c, int* a, int* b)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    c[i] = a[i] * b[i];
    __syncthreads();
    dataPerBlock[threadIdx.x] = c[i];

    int subtotal = 0;
    for (int k = 0; k < blockDim.x; k++)
        subtotal += dataPerBlock[k];

    c[blockIdx.x] = subtotal;
}

int main()
{
    cudaSetDevice(0);

    for (int i = 0; i < 8; i++)
    {
        a[i] = i + 1;
        b[i] = (i + 1) * 10;
    }

    PerElement_AtimesB << 2, 4 >>> (c, a, b);

    cudaDeviceSynchronize();

    printf("A is :{ %d", a[0]);
    for (int i = 1; i < 8; i++) {
        printf(",%d", a[i]);
    }
    printf(" }\n");
    printf("B is :{ %d", b[0]);
    for (int i = 1; i < 8; i++) {
        printf(",%d", b[i]);
    }
    printf(" }\n");
    printf("C is :{ %d", c[0]);
    for (int i = 1; i < 2; i++) {
        printf(",%d", c[i]);
    }
    printf(" }\n");
    int d = c[0] + c[1];
    printf("dot product of A & B is: %d", d);

    cudaDeviceReset();
    return 0;
}
```

Test data:

A	1	2	3	4	5	6	7	8
B	10	20	30	40	50	60	70	80

Sample output:

```
A is :{ 1,2,3,4,5,6,7,8 }  
B is :{ 10,20,30,40,50,60,70,80 }  
C is :{ 300,1740 }  
dot product of A & B is: 2040
```

Reflection:

The __sync thread function has been added into the kernel prior to the subtotal calculation this is to ensure all active threads reach this point before collapsing the values this is because a thread may still require access to c[0] and c[1] which are overwritten as a result of this subtotal calculation.

Question:

Task 2. Consider different thread configurations, for example, <<<1, 8>>>, <<<2, 4>>>, <<<4, 2>>> and observe if the above program can calculate the vector dot-product correctly. If not, analyse the issues and consider how to fix them.

Solution:

```
const int ArrayLength = 8;
const int threadsPerBlock = 2;
const int blocks = 4;

__device__ __managed__ int a[ArrayLength], b[ArrayLength], c[ArrayLength];
__shared__ int dataPerBlock[threadsPerBlock];

__global__ void PerElement_AtimesB(int* c, int* a, int* b)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    c[i] = a[i] * b[i];
    __syncthreads();
    dataPerBlock[threadIdx.x] = c[i];
    int subtotal = 0;
    for (int k = 0; k < blockDim.x; k++)
    {
        subtotal += dataPerBlock[k];
    }
    printf("subtotal is: %d", subtotal);
    c[blockIdx.x] = subtotal;
}

int main()
{
    cudaSetDevice(0);

    for (int i = 0; i < ArrayLength; i++)
    {
        a[i] = i + 1;
        b[i] = (i + 1) * 10;
    }

    PerElement_AtimesB << <blocks, threadsPerBlock >> > (c, a, b);

    cudaDeviceSynchronize();

    printf("A is :{ %d", a[0]);
    for (int i = 1; i < ArrayLength; i++) {
        printf(", %d", a[i]);
    }
    printf(" }\n");
    printf("B is :{ %d", b[0]);
    for (int i = 1; i < ArrayLength; i++) {
        printf(", %d", b[i]);
    }
    printf(" }\n");
    printf("C is :{ %d", c[0]);
    for (int i = 1; i < blocks; i++) {
        printf(", %d", c[i]);
    }
    printf(" }\n");
    int d = 0;
    for (int i = 0; i < blocks; i++){
        d += c[i];
    }
    printf("dot product of A & B is: %d", d);

    cudaDeviceReset();
    return 0;
}
```

Test data:

Sample output:

Reflection:

In its initial state the program cannot compile as it only handles up to 2 return blocks to change the code so that it can handle any combination of threads and blocks then adjustments need to be made to the shared variable this is done by adding a set of variables to define these at the head of the solution adjusting these allows for any size of array the only restriction is that the $\text{blocks} * \text{threadsPerBlock}$ must be greater than the `ArrayLength` variable

Metadata:

"Dot-Product", "thread", "c++", "GPU", "declared", "managed"

Further information:

N/A

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

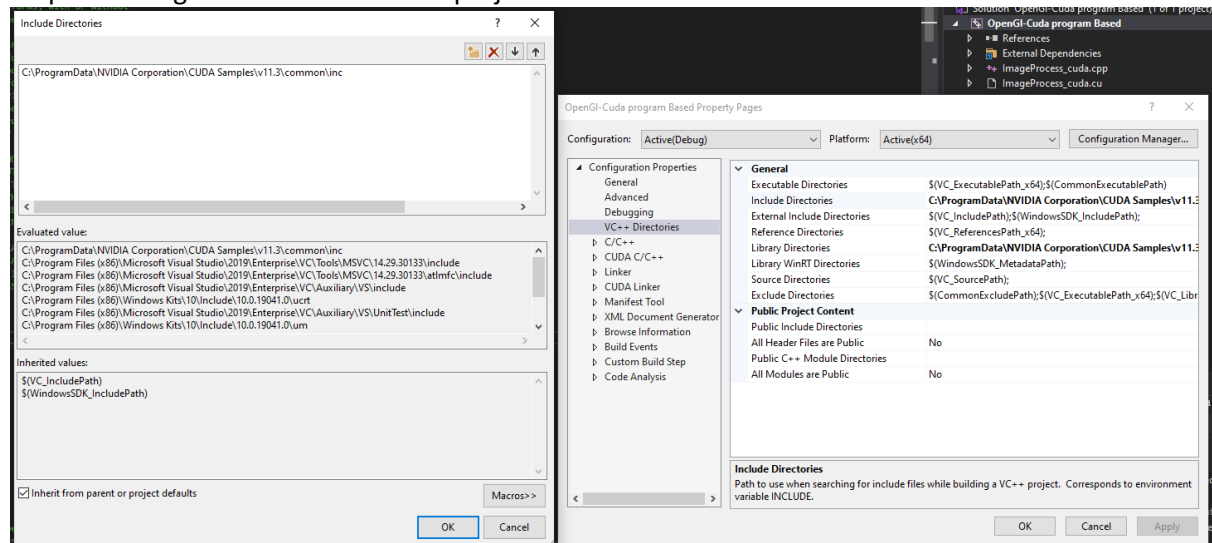
Question:

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

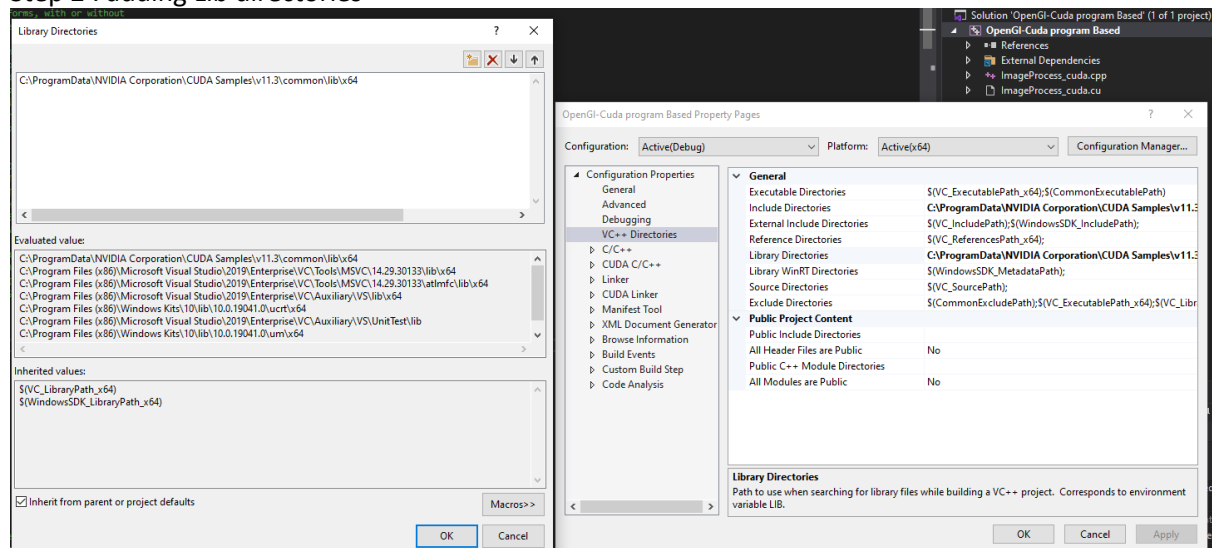
Solution:

No sample code to show

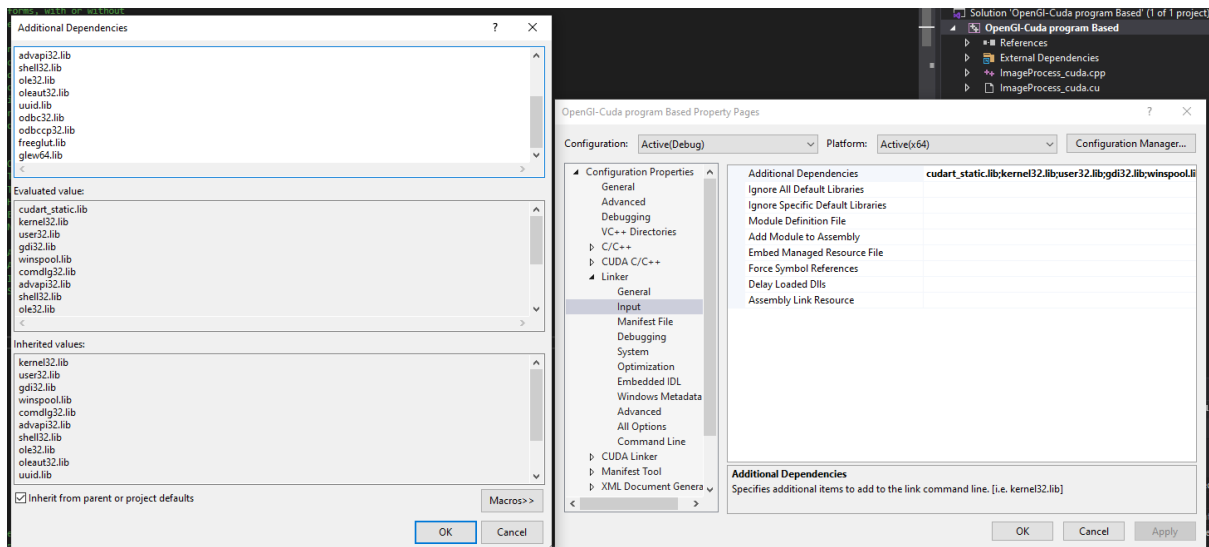
Step 1 : adding include directories to project



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 <lenna_bw.pgm> in ./
sdkFindFilePath <lenna_bw.pgm> in ./OpenGL-Cuda program Based_data_files/
sdkFindFilePath <lenna_bw.pgm> in ./common/
sdkFindFilePath <lenna_bw.pgm> in ./common/data/
sdkFindFilePath <lenna_bw.pgm> in ./data/
sdkFindFilePath <lenna_bw.pgm> in ./src/
sdkFindFilePath <lenna_bw.pgm> in ./src/OpenGL-Cuda program Based/data/
sdkFindFilePath <lenna_bw.pgm> in ./inc/
sdkFindFilePath <lenna_bw.pgm> in ./0_Simple/
sdkFindFilePath <lenna_bw.pgm> in ./1_Uutilities/
sdkFindFilePath <lenna_bw.pgm> in ./2_Graphics/
sdkFindFilePath <lenna_bw.pgm> in ./3_Imaging/
sdkFindFilePath <lenna_bw.pgm> in ./4_Finance/
sdkFindFilePath <lenna_bw.pgm> in ./5_Simulations/
sdkFindFilePath <lenna_bw.pgm> in ./6_Advanced/
sdkFindFilePath <lenna_bw.pgm> in ./7_CUDAlibraries/
sdkFindFilePath <lenna_bw.pgm> in ./8_Android/
sdkFindFilePath <lenna_bw.pgm> in ./samples/
sdkFindFilePath <lenna_bw.pgm> in ./0_Simple/OpenGL-Cuda program Based/data/
sdkFindFilePath <lenna_bw.pgm> in ./1_Uutilities/OpenGL-Cuda program Based/data/
sdkFindFilePath <lenna_bw.pgm> in ./2_Graphics/OpenGL-Cuda program Based/data/
sdkFindFilePath <lenna_bw.pgm> in ./3_Imaging/OpenGL-Cuda program Based/data/
sdkFindFilePath <lenna_bw.pgm> in ./4_Finance/OpenGL-Cuda program Based/data/
sdkFindFilePath <lenna_bw.pgm> in ./5_Simulations/OpenGL-Cuda program Based/data/
sdkFindFilePath <lenna_bw.pgm> in ./6_Advanced/OpenGL-Cuda program Based/data/
sdkFindFilePath <lenna_bw.pgm> in ./7_CUDAlibraries/OpenGL-Cuda program Based/data/
sdkFindFilePath <lenna_bw.pgm> in ./7_CUDAlibraries/OpenGL-Cuda program Based/data/
sdkFindFilePath <lenna_bw.pgm> in ./
Loaded 'lenna_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);
```

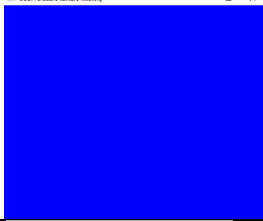
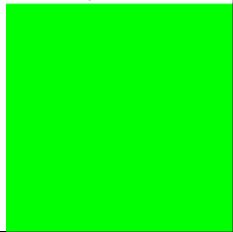


Running each of these one at a time and commenting out the other to display the different resulting outcomes

```
__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
<code>d_output[i] = make_uchar4(0xff, 0, 0, 0);</code>	
<code>d_output[i] = make_uchar4(0, 0xff, 0, 0);</code>	
<code>d_output[i] = make_uchar4(0, 0, 0xff, 0);</code>	
<code>d_output[i] = make_uchar4(0, 0, 0xff, 0);</code>	

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 };

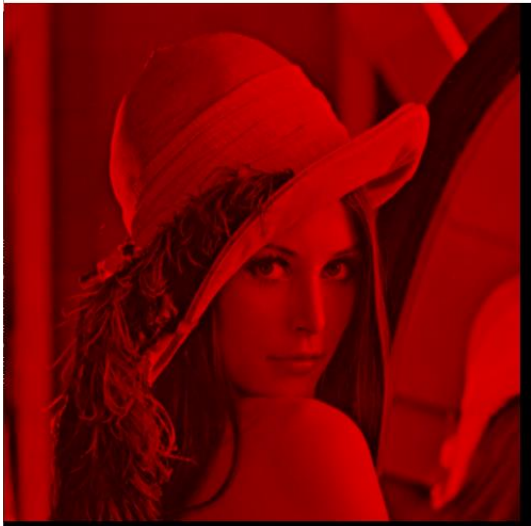
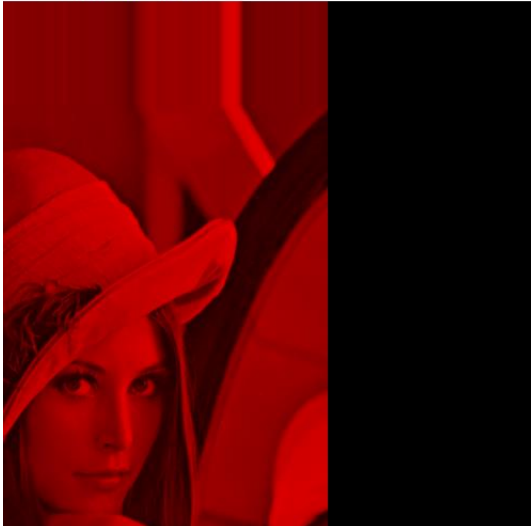
    T:
        x += T.x;
        y += T.y;
        ty;

    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
<code>float2 T = { 20, 10 };</code>	
<code>float2 T = { 200, -100 };</code>	

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

- 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);
    }
}
```

- 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);
    }
}
```

- 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
<code>float2 S = { 1.2, 0.5 };</code>	
<code>float2 S = { 2, -0.5 };</code>	

Reflection:

The image has been translated by scaling 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:

$$R = \begin{pmatrix} \cos(\text{angle}) & -\sin(\text{angle}) \\ \sin(\text{angle}) & \cos(\text{angle}) \end{pmatrix}$$
$$\text{float rx} = x * \cos(\text{angle}) - y * \sin(\text{angle});$$
$$\text{float ry} = x * \sin(\text{angle}) + y * \cos(\text{angle});$$

```
__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;

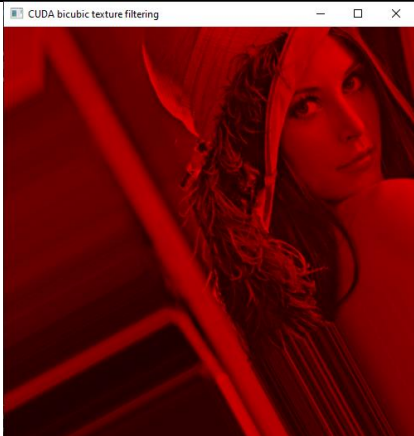
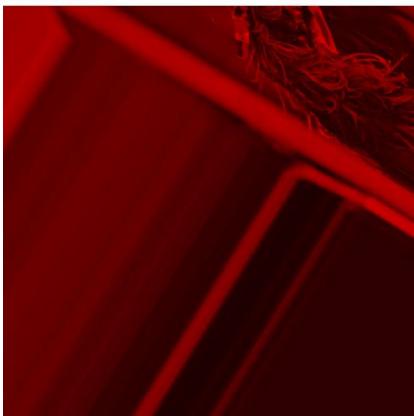
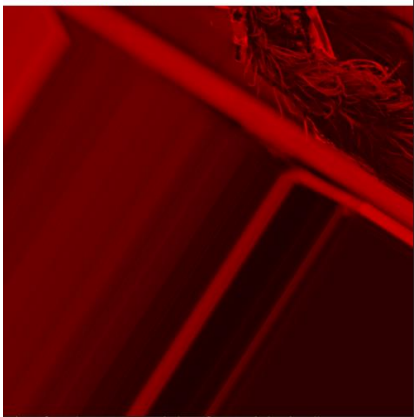

    float rx = x * cos(angle) - y * sin(angle);
    float ry = x * sin(angle) + y * cos(angle);

    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
<code>float angle = 0.5;</code>	
<code>float angle = 45;</code>	
<code>float angle = 1;</code>	
<code>float angle = -0.5;</code>	

Reflection:

The image has been around the origin which is the top left however I am unsure what the angle 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:

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

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

```
__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, rgbaTexImage);

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

Sample output:

Translation	result
<pre>scale = 1.3, cx = 35, cy = 0;</pre>	
<pre>scale = 0.5, cx = 35, cy = 1;</pre>	

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;

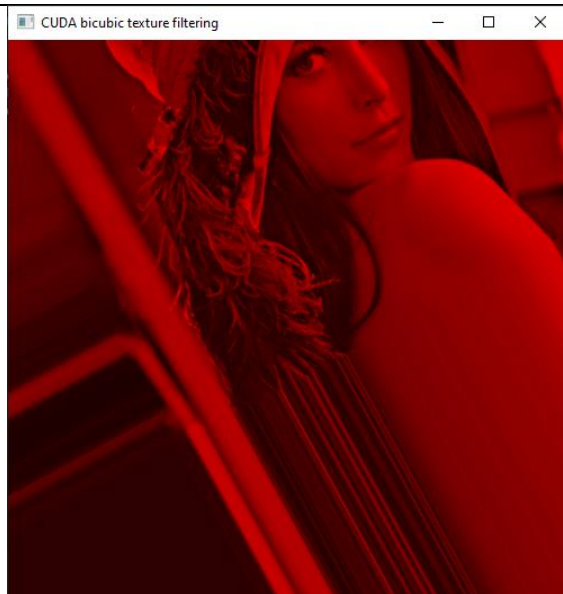
    if ((x < width) && (y < height)) {
        // write output color

        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

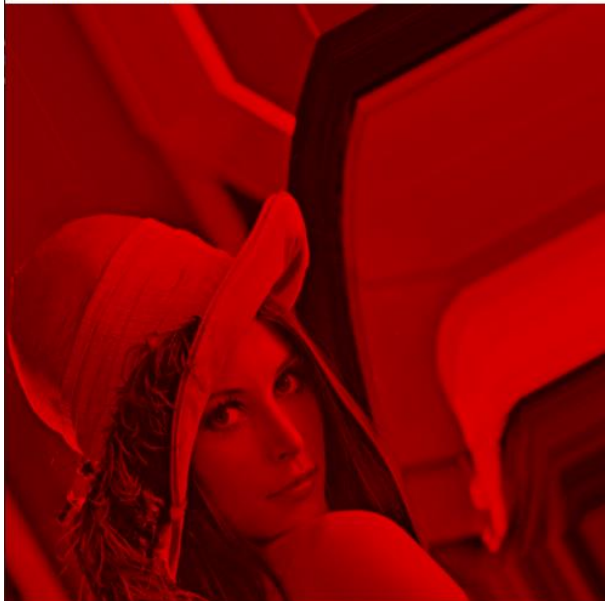
    float u = (x - cx) * scale + cx + tx;
    float v = (y - cy) * scale + cy + ty;

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

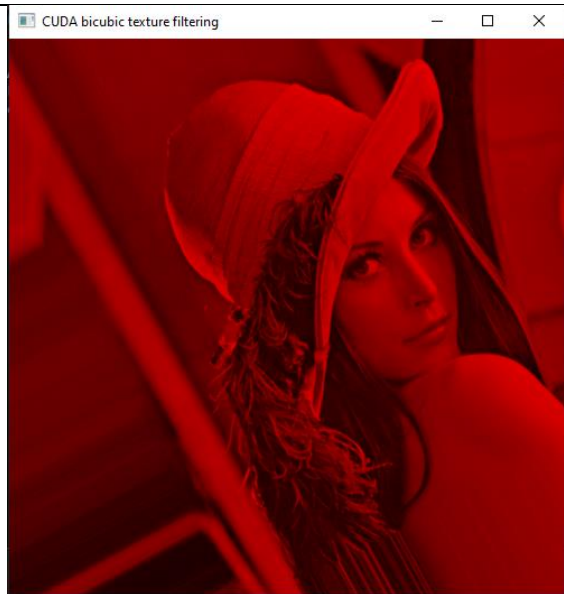
    if ((x < width) && (y < height)) {
        // write output color

        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.

600086 Lab Book

Week 5 – Lab B

Date: 3rd March 2022

Exercise 1. A simple matrix multiplication program in CUDA using one thread block

Question1 : implement a solution in C++ to the problem

Solution:

1. Created a C++ solution that iterates through the required combinations. This is done using a series of for loops. Taking the dimensions of the final array C and iterating through the x and y values as shown in the code below.

```
void ConcurrentMatrixMultiplication(int* c, int* a, int* b, int common, int W, int H)
{
    for (int x = 0; x < W; x++)
    {
        for (int y = 0; y < H; y++)
        {
            int ARow = y * common;
            int BColumn = x;
            int CIndex = x + (y * W);

            printf("Thread ID: (%d,%d)\n AIndex: %d\n BIndex: %d\n CIndex: %d\n", x, y, ARow, BColumn, CIndex);

            for (int i = 0; i < common; i++)
            {
                c[CIndex] += (a[ARow + i]) * (b[i * W + BColumn]);
            }
        }
    }
}
```

The commented-out printf for debugging purposes and produces the following output

```
Thread ID: (1,1)
{
  AIndex: 2
  BIndex: 1
  CIndex: 3
}
```

This allowed me to monitor the index values for finding the various matrix index value based on the threadIdx values. And ensure the offset values matched the expected coordinates.

Test data:

$$\text{Matrix } A \begin{Bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{Bmatrix} \text{Matrix } B \begin{Bmatrix} 10 & 20 \\ 30 & 40 \\ 50 & 60 \end{Bmatrix}$$

Sample output:

```
Array C = {220,280,490,640,760,1000,1030,1360}
```

Question2 : implement a solution using GPU processing to solve the problem

Solution:

1. Modified the Add kernel Function to multiply the matrixes as follows

```
// represents the Common Dimension between the two arrays
__global__ void OneBlockMatrixMultiplication(int* c, int* a, int* b, int common, int cW)
{
    int ARow = threadIdx.y * common;
    int BColumn = threadIdx.x;
    int CIndex = threadIdx.x + (threadIdx.y * cW);

    //printf("Thread ID: (%d,%d)\n\n AIndex: %d\n BIndex: %d\n CIndex: %d\n\n", threadIdx.x, threadIdx.y, ARow, BColumn, CIndex);

    for (int i = 0; i < common; i++)
    {
        c[CIndex] += (a[ARow + i]) * (b[i * cW + BColumn]);
    }
}
```

2. Added two new passed in variables: "common" that represents the common dimension of the matrices to make calculating the indexes of the arrays easier to manage and "cW" which represents the width of the B array to allow for offset calculations.

the for loop iterates through and calculates the dot product of the current row and column this could be extracted into a further kernel function to speed up the process.

The commented-out printf for debugging purposes and produces the following output

```
Thread ID: (1,1)
{
  AIndex: 2
  BIndex: 1
  CIndex: 3
}
```

This allowed me to monitor the index values for finding the various matrix index value based on the threadIdx values. And ensure the offset values matched the expected coordinates.

Test data:

$$\text{Matrix } A \begin{Bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{Bmatrix} \text{Matrix } B \begin{Bmatrix} 10 & 20 \\ 30 & 40 \\ 50 & 60 \end{Bmatrix}$$

Sample output:

```
Array A = {1,2,3,4,5,6,7,8,9,10,11,12}
Array B = {10,20,30,40,50,60}
Array C = {30,70,50,110,70,150,90,190}
```

Reflection:

This was fairly simple the most difficult part was collapsing the Thread in the right way for the X and Y values as I kept confusing the two and doing them the wrong way round. However as I had looped via x and y dimensions for the concurrent method modifying the code to run on GPU in parallel allowed me to sub in the threadIdx.x and threadIdx.y for my x and y variables.

Metadata:

Further information:

Is it possible to launch a kernel from a kernel if so this could improve the process?

Exercise 2. Compare the performance of the CUDA solution against the CPU solution

Question1 : Running performance test on a range of square arrays for CPU solution and GPU solution comparing the results

Solution:

N/A

Test data:

Matrix A = Matrix B

Matrix B will be a square matrix of the following sizes

(8*8),(32*32),(256*256),(512*512),(1024*1024)

Sample output:

Matrix dimensions	CPU	GPU
8 x 8	Execution of kernel: 0.018720ms	Execution of kernel: 0.082272ms
32 x 32	Execution of kernel: 0.020480ms	Execution of kernel: 0.114464ms
256 x 256	Execution of kernel: 44.437504ms	Execution of kernel: 0.018784ms
512 x 512	Execution of kernel: 417.548157ms	Execution of kernel: 0.055744ms
1024 x 1024	Execution of kernel: 3544.644775ms	Execution of kernel: 0.020800ms

Reflection:

The CPU only solution seemed faster initially however it quickly began to take more and more time whereas the GPU solution kept a fairly consistent execution time the primary bottleneck was printing the results to the screen however another bottleneck was the iteration on each thread this could be replaced by a nested kernel call however I was unsure if this was possible.

Metadata:

Further information:

Is it possible to launch a kernel from a kernel if so this could improve the process?

600086 Lab Book

Week 2 – Lab 6 A simple CUDA ray caster

Date: 10th Mar 2022

Exercise 1. Set up a virtual canvas and draw on it an image in CUDA

Question1 :

Modify the first three values shown in `make_uchar4()` in the following line of code to draw an image of different colours, say, a green image, a grey image.

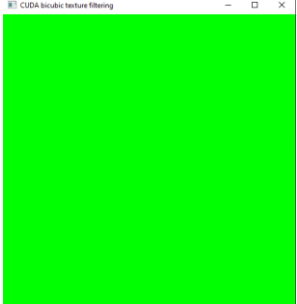
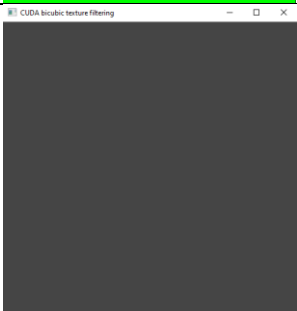

Solution:

```
if ((x < width) && (y < height)) {  
    d_output[i] = make_uchar4(0, 0xFF, 0, 0);  
}
```

Test data:

N/A

Sample output:

Input	expectation	Output
(0, 0xFF, 0, 0)	Green	
(0x45, 0xF45, 0x45, 0)	Grey	
(0, 0xFF, 0, 0)	Fuchsia	

Exercise 2. Drawing a checkboard in CUDA

Question2: implement a solution using GPU processing to solve the problem

Solution:

1. Edit the `d_render()` method to draw a checkboard

```
__global__ void d_render(uchar4* d_output, uint width, uint height) {
    uint x = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
    uint y = __umul24(blockIdx.y, blockDim.y) + threadIdx.y;
    uint i = __umul24(y, width) + x;
    uint c = (((x & 0x8) == 0) ^ ((y & 0x8) == 0));
    if ((x < width) && (y < height)) {
        d_output[i] = make_uchar4(c, c, c * 0xff, 0);
    }
}
```

Created a new variable `c` which is governed by the `x` and `y` position of the pixel and applied a colour mask to it in the `make_uchar4()` to make the odd segments red. See Sample output ref1 for result

2. Modify the code to draw a checkboard with much larger red-blocks

```
__global__ void d_render(uchar4* d_output, uint width, uint height) {
    uint x = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
    uint y = __umul24(blockIdx.y, blockDim.y) + threadIdx.y;
    uint i = __umul24(y, width) + x;
    uint c = (((x & 0x80) == 0) ^ ((y & 0x80) == 0));
    if ((x < width) && (y < height)) {
        d_output[i] = make_uchar4(c, c, c * 0xff, 0);
    }
}
```

By increasing the value that `x` and `y` are multiplied by when calculating `c` the size of the squares in the grid can be modified I increased this to `0x80`. see Sample output Ref 2 for result.

3. Further modify your code to draw a red disc in the middle of the image of a red disc:

```
__global__ void d_render(uchar4* d_output, uint width, uint height) {
    uint x = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
    uint y = __umul24(blockIdx.y, blockDim.y) + threadIdx.y;
    uint i = __umul24(y, width) + x;
    uint c = 255;
    uint r = 45;
    //c = (((x & 0x80) == 0) ^ ((y & 0x80) == 0));
    if ((x < width) && (y < height))
    {
        float dist = sqrtf((x - (width / 2)) * (x - (width / 2)) + (y - (height / 2)) * (y - (height / 2)));
        if(dist < r)
        {
            d_output[i] = make_uchar4(0x00, 0x00, 0xff, 0);
        }
        else
        {
            d_output[i] = make_uchar4(0x66, 0x99, 0x00, 0);
        }
    }
}
```

I added in a check to see if the coordinate distance of the pixel is within the range `r` and if so colour it red if not colour it teal. See Sample output Ref 3 for the result.

4. Redraw the image based on pixel coordinates defined in float type variables in $[-1, 1] \times [-1, 1]$

```
__global__ void d_render(uchar4* d_output, uint width, uint height) {
    uint x = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
    uint y = __umul24(blockIdx.y, blockDim.y) + threadIdx.y;
    uint i = __umul24(y, width) + x;
    float u = x / (float)width;
    float v = y / (float)height;
    u = 2.0 * u - 1.0;
    v = -(2.0 * v - 1.0);
    //scale u by aspect ratio
    u *= width / (float)height;

    uint c = 255;
    float r = 0.5;
    if ((x < width) && (y < height))
    {
        float dist = sqrtf(powf(u - (0), 2) + powf(v - (0), 2));

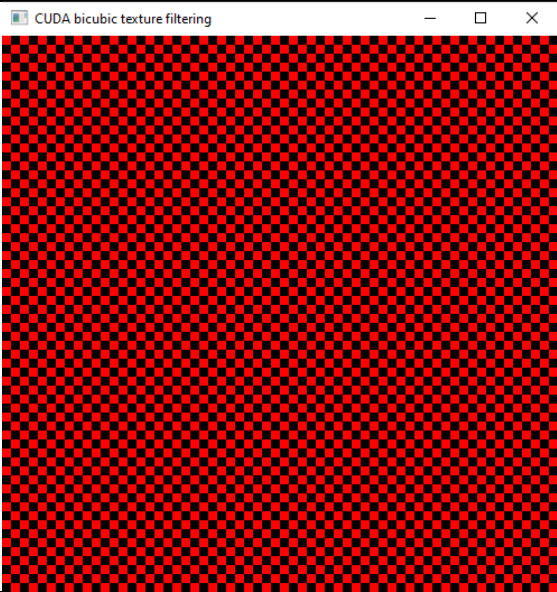
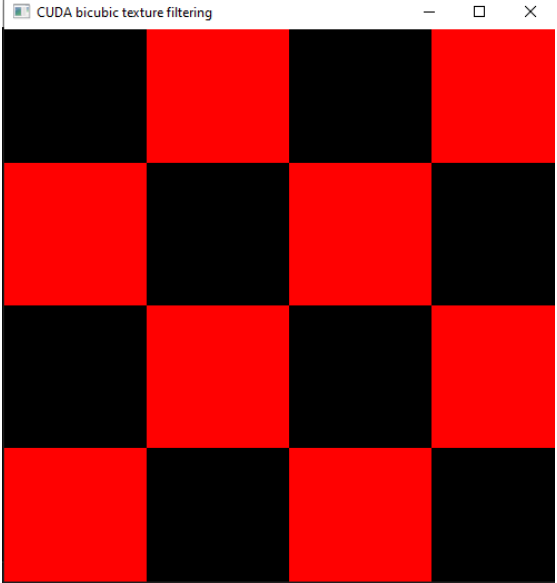
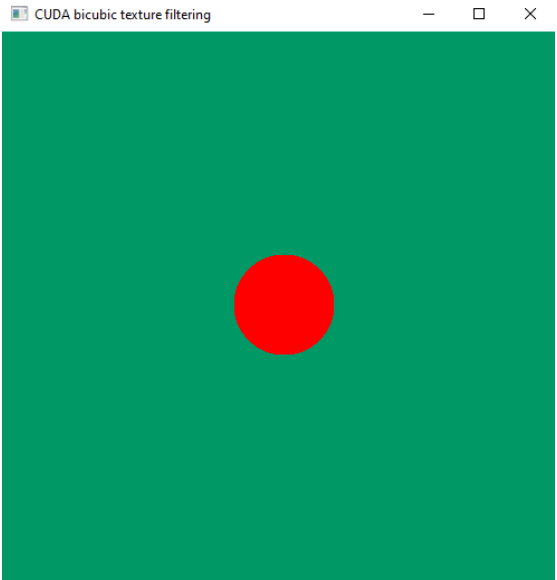
        if(dist < r)
        {
            d_output[i] = make_uchar4(0x00, 0x00, 0xff, 0);
        }
        else
        {
            d_output[i] = make_uchar4(0x66, 0x99, 0x00, 0);
        }
    }
}
```

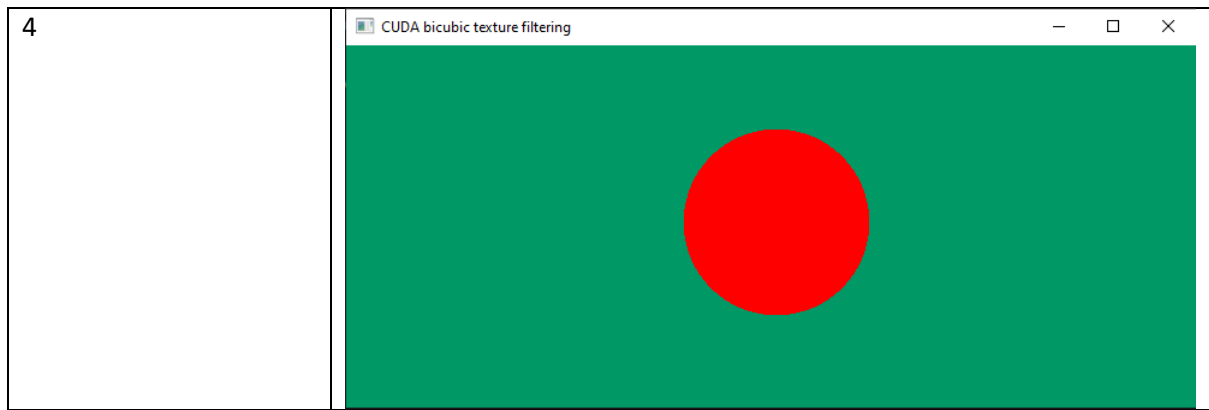
I Added in a translation for the pixel location represented by u and v and then added a scale translation to ensure the resultant image matched the aspect ratio of the window to prevent distortion. See Sample Output Ref 4 for the result.

Test data:

N/A

Sample output:

REF	Output
1	
2	
3	



Reflection:

This task seemed fairly perfunctory, but was very interesting seeing how shapes can be drawn to the screen using vectors,

Metadata:

Further information:

is this similar to how vector graphics are created?

Exercise 3. Drawing the Mandelbrot and Julia Sets.

Question1: modify the previous code in order to draw Mandelbrot and Julia sets.

Solution:

1. Modify the code to draw a Mandelbrot set

```
__global__ void d_render(uchar4* d_output, uint width, uint height) {
    uint x = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
    uint y = __umul24(blockIdx.y, blockDim.y) + threadIdx.y;
    uint i = __umul24(y, width) + x;
    float u = x / (float)width;
    float v = y / (float)height;
    u = 2.0 * u - 1.0;
    v = -(2.0 * v - 1.0);
    //scale u by aspect ratio
    u *= width / (float)height;
    u *= 2.0;
    v *= 2.0;
    float2 z = { u,v };
    float2 T = z;
    float r = 0.0;
    float c = 1.0;
    for (int i = 0; i < 30; i++)
    {
        z = { z.x * z.x - z.y * z.y, 2.0f * z.x * z.y, };
        z += T;
        r = sqrtf(z.x * z.x + z.y * z.y);
        if (r > 5.0)
        {
            c = 0.0;
            break;
        }
    }
    if ((x < width) && (y < height)) {
        d_output[i] = make_uchar4(c*0x00, c*0x00, c*0xff, 0);
    }
}
```

Added in a for loop that iteratively validates whether the current pixel is not within the Mandelbrot set and leaves the loop early if this is the case setting the c value to zero so the pixel will be black. See sample output ref 1 for resultant image.

2. Modify the code to draw a Julia set

By changing the Vector T so that it is not the start coordinates then we can create Julia sets as there are an infinite number of Julia sets I have created 3 using the Vector values for T in the Test Data section the results can be seen in Sample output 2,3 and 4 respectively

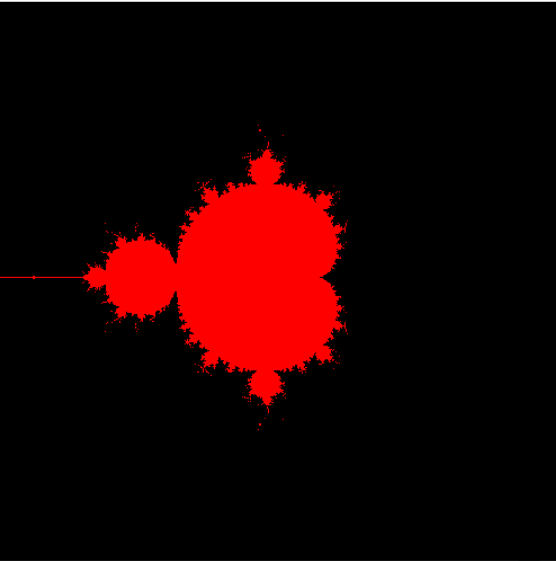
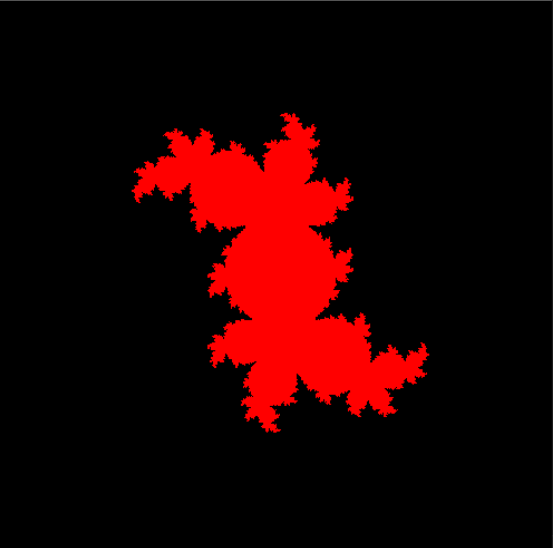
Test data:

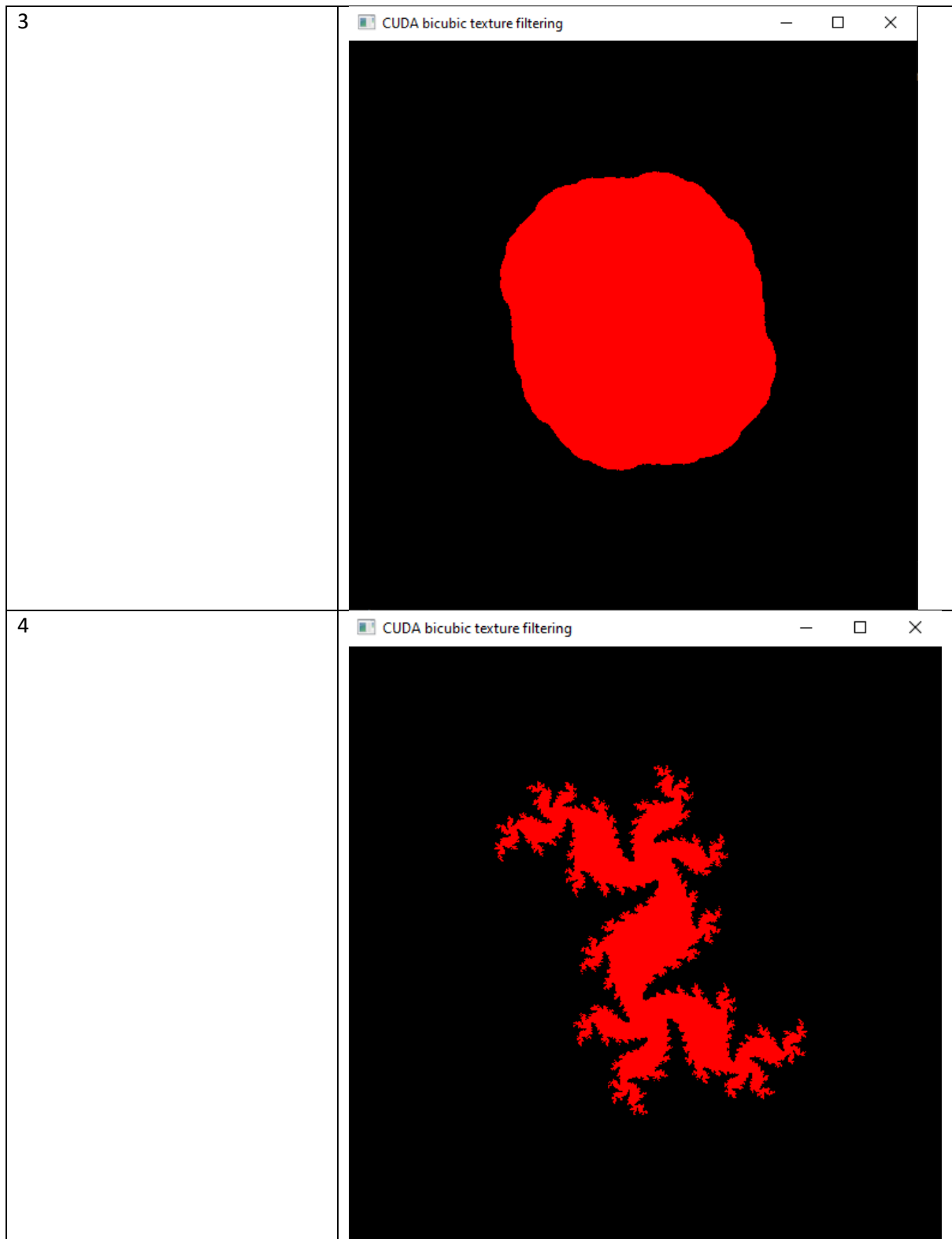
T = {0.25, 0.5}

T = {0.1, 0.1}

T = {0.3, 0.5}

Sample output:

ref	Output
1	
2	



Reflection:

Adjusting the x value of the T vector makes the Julia set pattern have deeper grooves whereas the y value seems to cause the pattern to have softer edges effectively smoothing the shape.

Metadata:

600086 Lab Book

Week 7 – Lab 6 A simple CUDA ray caster

Date: 24th Mar 2022

Exercise 1. Drawing based on a canvas of size $[-1, 1] \times [-1, 1]$

Question1: implement a solution using GPU processing to solve the problem

Solution:

1. Draw the image based on pixel coordinates defined in float type variables in $[-1, 1] \times [-1, 1]$

```
__global__ void d_render(uchar4* d_output, uint width, uint height) {
    uint x = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
    uint y = __umul24(blockIdx.y, blockDim.y) + threadIdx.y;
    uint i = __umul24(y, width) + x;
    float u = x / (float)width;
    float v = y / (float)height;
    u = 2.0 * u - 1.0;
    v = -(2.0 * v - 1.0);
    //scale u by aspect ratio
    u *= width / (float)height;

    uint c = 255;
    float r = 0.5;
    if ((x < width) && (y < height))
    {
        float dist = sqrtf(powf(u - (0), 2) + powf(v - (0), 2));

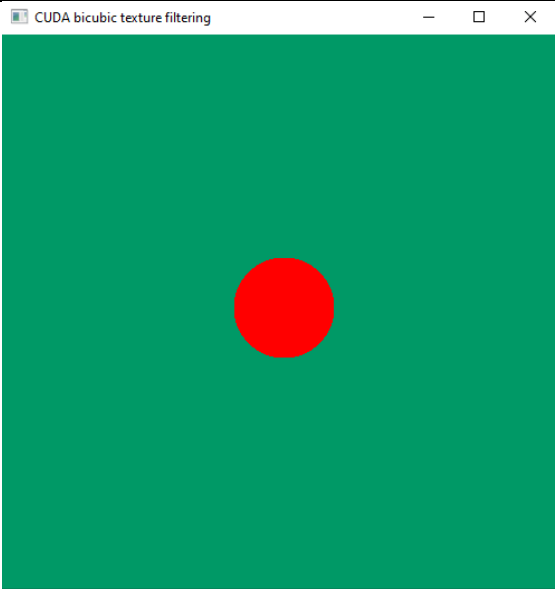
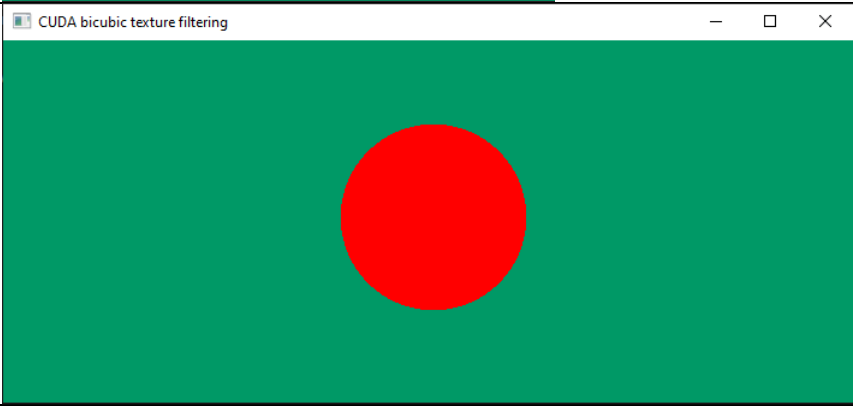
        if(dist < r)
        {
            d_output[i] = make_uchar4(0x00, 0x00, 0xff, 0);
        }
        else
        {
            d_output[i] = make_uchar4(0x66, 0x99, 0x00, 0);
        }
    }
}
```

Added in a translation for the pixel location represented by u and v and then added a scale translation to ensure the resultant image matched the aspect ratio of the window to prevent distortion. See Sample Output Ref 1 and 2 for the results.

Test data:

N/A

Sample output:

REF	Output
2	
2	

Reflection:

none

Metadata:

Further information:

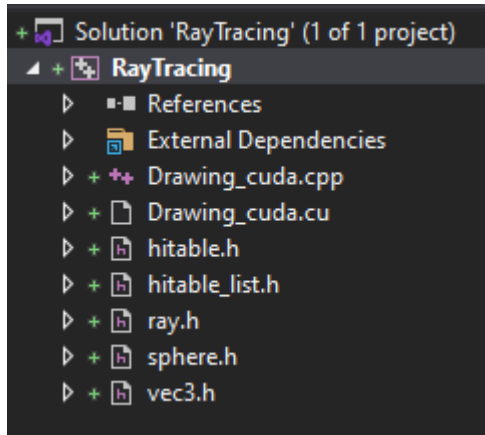
none

Exercise 2. Write a simple ray caster

Question1: implement ray casting based on raycasting in a weekend repo

Solution:

1. Add the necessary header files to the project



2. Change the variable names in the ray class to make them more meaningful for the current implementation

```
class ray
{
public:
    __device__ ray() {}
    __device__ ray(const vec3& a, const vec3& b) { O = a; Dir = b; }
    __device__ vec3 origin() const { return O; }
    __device__ vec3 direction() const { return Dir; }
    __device__ vec3 point_at_parameter(float t) const { return O + t * Dir; }

    vec3 O;
    vec3 Dir;
};
```

Changed the A and B variables to be O for origin and Dir for direction.

3. Implement the following functions cuda_check_error, castRay, create_world and free_world

```
#define checkCudaErrors(val) check_cuda( (val), #val, __FILE__, __LINE__ )
void check_cuda(cudaError_t result, char const* const func, const char* const file, int const line) {
    if (result) {
        std::cerr << "CUDA error = " << static_cast<unsigned int>(result) << " at " <<
            file << ":" << line << " '" << func << "' \n";
        // Make sure we call CUDA Device Reset before exiting
        cudaDeviceReset();
        exit(99);
    }
}
```

```

__device__ vec3 castRay(const ray& r, hitable** world) {
    hit_record rec;
    if ((*world)->hit(r, 0.0, FLT_MAX, rec)) {
        return 0.5f * vec3(rec.normal.x() + 1.0f, rec.normal.y() + 1.0f, rec.normal.z() + 1.0f);
    }
    else {
        vec3 unit_direction = unit_vector(r.direction());
        float t = 0.5f * (unit_direction.y() + 1.0f);
        return (1.0f - t) * vec3(1.0, 1.0, 1.0) + t * vec3(0.5, 0.7, 1.0);
    }
}

__global__ void create_world(hitable** d_list, hitable** d_world) {
    if (threadIdx.x == 0 && blockIdx.x == 0) {
        *(d_list) = new sphere(vec3(0, 0, -1), 0.5);
        *(d_list + 1) = new sphere(vec3(0, -100.5, -1), 100);
        *d_world = new hitable_list(d_list, 2);
    }
}

__global__ void free_world(hitable** d_list, hitable** d_world) {
    delete* (d_list);
    delete* (d_list + 1);
    delete* d_world;
}

```

4. Modify the d_render method so that it will raycast and render the image

```

__global__ void d_render(uchar4* d_output, uint width, uint height, hitable** d_world)
{
    uint x = blockIdx.x * blockDim.x + threadIdx.x;
    uint y = blockIdx.y * blockDim.y + threadIdx.y;
    uint i = y * width + x;

    float u = x / (float)width;
    float v = y / (float)height;

    u = 2.0 * u - 1.0;
    v = -(2.0 * v - 1.0);
    //scale u by aspect ratio
    u *= width / (float)height;

    vec3 eye = vec3(0, 0.5, 1.5);
    float distFromEyeToImg = 1.0;
    if ((x < width) && (y < height))
    {
        vec3 pixelPos = vec3(u, v, eye.z() - distFromEyeToImg);
        ray r;
        r.O = eye;
        r.Dir = pixelPos - eye;

        vec3 col = castRay(r, d_world);
        float red = col.x();
        float green = col.y();
        float blue = col.z();
        d_output[i] = make_uchar4(red * 255, green * 255, blue * 255, 0);
    }
}

```

5. Modify the render() method so that it will create a sphere and pass it into the d_render method for casting

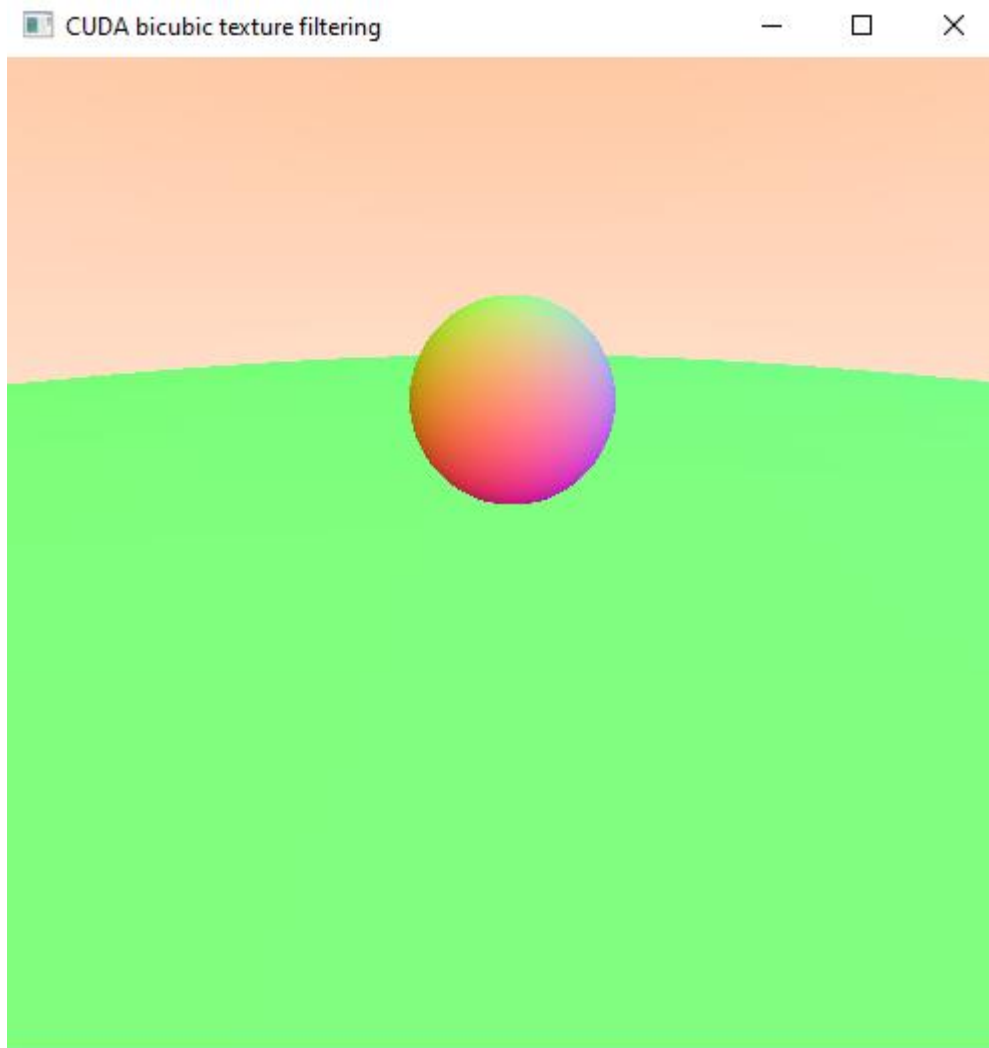
```
// render image using CUDA
extern "C"
{
    void render(int width, int height, dim3 blockSize, dim3 gridSize, uchar4 * output)
    {
        hitable **d_list;
        checkCudaErrors(cudaMalloc((void **)&d_list, 2 * sizeof(hitable*)));
        hitable **d_world;
        checkCudaErrors(cudaMalloc((void **)&d_world, sizeof(hitable*)));
        create_world << <1, 1 >> > (d_list, d_world);
        checkCudaErrors(cudaGetLastError());
        checkCudaErrors(cudaDeviceSynchronize());
        d_render << <gridSize, blockSize >> > (output, width, height, d_world);
        getLastCudaError("kernel failed");
    }
}
```

Running the solution at this point renders the image shown in the sample out put section.

Test data:

none

Sample output:



Reflection:

The task above was fairly easy to follow unsure why my red and blue values seem to have switched making my sky tinged red rather than the blue in the example.

Metadata:

Exercise 3. Adding multiple spheres to the ray caster

Question1: modify the previous code to render 10 spheres on the screen

Solution:

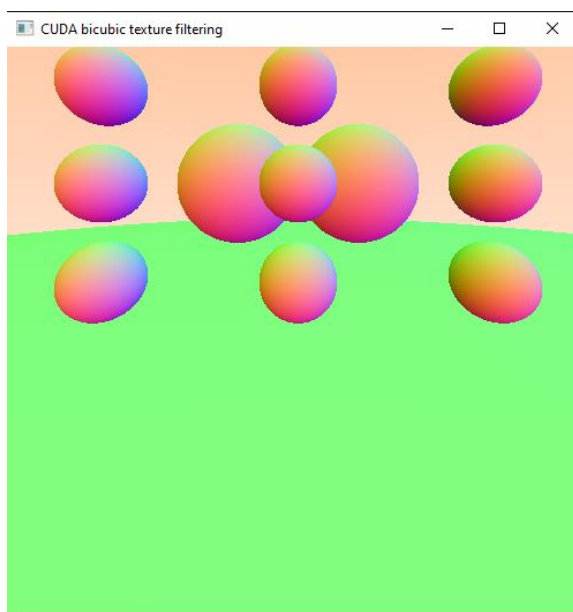
```
__global__ void create_world(hitable** d_list, hitable** d_world) {  
    if (threadIdx.x == 0 && blockIdx.x == 0) {  
        *(d_list)  
        = new sphere(vec3(1, 1, 0), 0.2);  
        *(d_list + 1)  
        = new sphere(vec3(1, 0.5, 0), 0.2);  
        *(d_list + 2)  
        = new sphere(vec3(1, 0, 0), 0.2);  
        *(d_list + 3)  
        = new sphere(vec3(0, 1, 0), 0.2);  
        *(d_list + 4)  
        = new sphere(vec3(0, 0.5, 0), 0.2);  
        *(d_list + 5)  
        = new sphere(vec3(0, 0, 0), 0.2);  
        *(d_list + 6)  
        = new sphere(vec3(-1, 1, 0), 0.2);  
        *(d_list + 7)  
        = new sphere(vec3(-1, 0.5, 0), 0.2);  
        *(d_list + 8)  
        = new sphere(vec3(-1, 0, 0), 0.2);  
        *(d_list + 9)  
        = new sphere(vec3(-0.5, 0.5, -1), 0.5);  
        *(d_list + 10)  
        = new sphere(vec3(0.5, 0.5, -1), 0.5);  
        *(d_list + 11)  
        = new sphere(vec3(0, -100.5, -1), 100);  
  
        *d_world = new hitable_list(d_list, 12);  
    }  
}
```

Added in the 10 extra spheres to form a 3*3 grid of spheres with two larger spheres in the background further back. See sample output for resulting render.

Test data:

none

Sample output:



600086 Lab Book

Week 8 – Lab 8 A simple Particle animation in CUDA

Date: 24th Mar 2022

Exercise 1. Draw a box without front wall.

Question1: adjust the code from lab 7 so that you have an open box drawn to the screen

Solution:

I added in a set of 5 spheres that will make up the box they are very large making the curvature barely noticeable at our current scale.

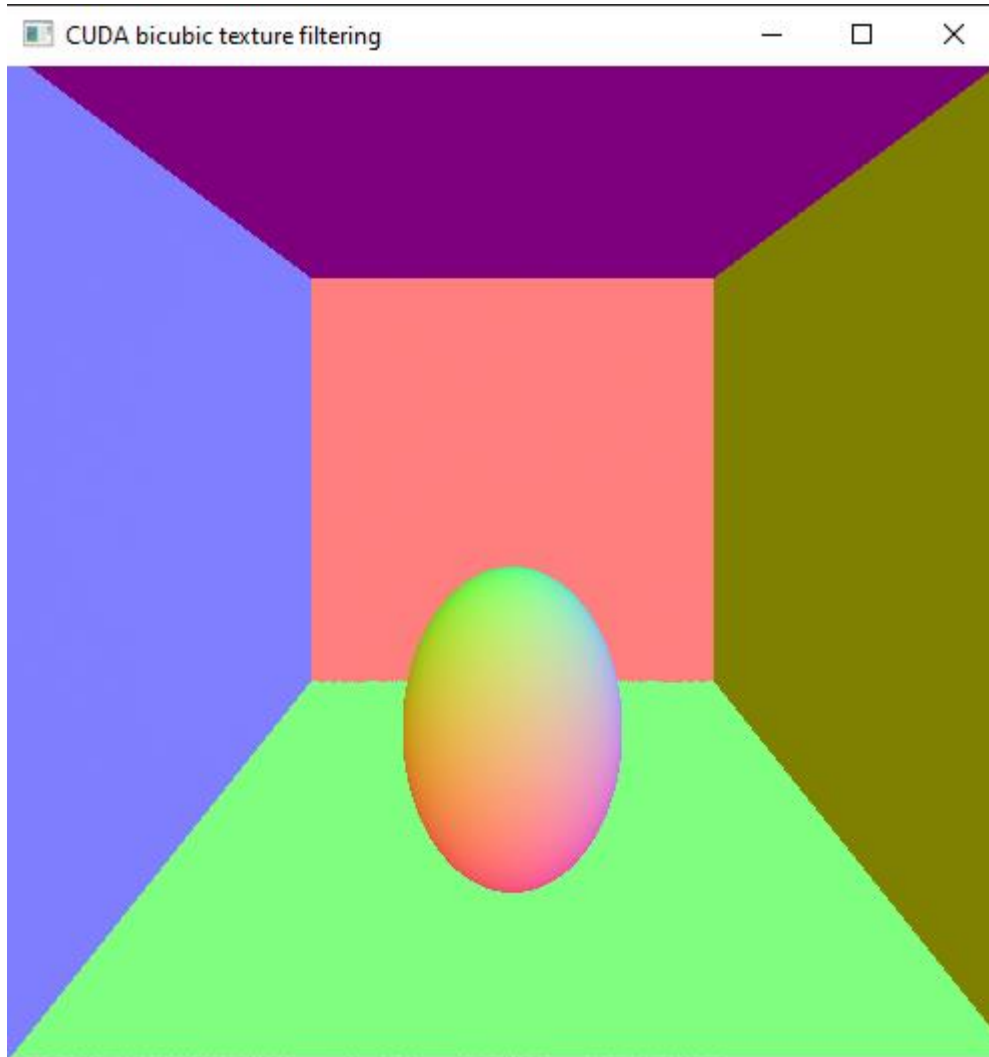
```
__global__ void create_world(hitable** d_list, hitable** d_world) {
    if (threadIdx.x == 0 && blockIdx.x == 0) {
        //Create the walls
        *(d_list + 0) = new sphere(vec3(-10002.0, 0, -3), 10000);
        *(d_list + 1) = new sphere(vec3(10002.0, 0, -3), 10000);
        *(d_list + 2) = new sphere(vec3(0, -10002.0, -3), 10000);
        *(d_list + 3) = new sphere(vec3(0, 10002.0, -3), 10000);
        *(d_list + 4) = new sphere(vec3(0, 0, -10001.0), 10000);
        //Create the balls
        *(d_list + 5) = new sphere(vec3(0, 0, 1), 0.2);

        *d_world = new hitable_list(d_list, 6);
    }
}
```

Test data:

N/A

Sample output:



Reflection:

none

Metadata:

Further information:

Exercise 2. Free motion animation

Question1: implement code to allow the particle to rotate about the centre of the box

Solution:

Added a device wide global variable named tick

```
__device__ static int ticks = 1;
```

Changed the sphere to be drawn to the following

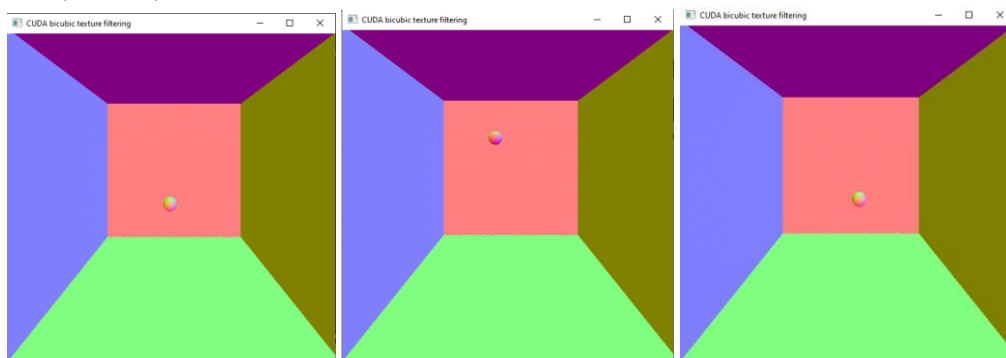
```
//Create the balls  
*(d_list + 5) = new sphere(vec3(cos(0.01 * (float)ticks++), sin(0.01 * (float)ticks++), -1), 0.2);
```

This moves the ball every time the image is rendered due to the tick variable being incremented

Test data:

n/a

Sample output:



Reflection:

This was fairly perfunctory and following the lab made sense

Metadata:

Exercise 3. Ball-box walls collision animation

Questions:

1. Modify the previous code to give the sphere a velocity and bounce off of the walls of the box
2. Implement a code change that will make the ball change colour after a collision.
3. Implement code change to allow multiple balls to move at the same time;

Solution:

1. I first defined some global variables for the sphere to track its movement

```
__device__ static int ticks = 1;
__device__ static double x_position = 0.0;
__device__ static double y_position = 0.0;
__device__ static double x_velocity = 0.02;
__device__ static double y_velocity = 0.03;
```

I then modified the create world kernel to move the ball and check for collisions with the walls

```
__global__ void create_world(hitable** d_list, hitable** d_world) {
    if (threadIdx.x == 0 && blockIdx.x == 0) {
        //define vectors
        vec3 left = vec3(-10002.0, 0, 0);
        vec3 right = vec3(10002.0, 0, 0);
        vec3 bottom = vec3(0, -10002.0, 0);
        vec3 top = vec3(0, 10002.0, 0);
        vec3 back = vec3(0, 0, -10001.0);
        int wall_size = 10000;
        //Create the walls
        *(d_list + 0) = new sphere(left, wall_size);
        *(d_list + 1) = new sphere(right, wall_size);
        *(d_list + 2) = new sphere(bottom, wall_size);
        *(d_list + 3) = new sphere(top, wall_size);
        *(d_list + 4) = new sphere(back, wall_size);
        //Create the balls
        *(d_list + 5) = new sphere(vec3(x_position += x_velocity, y_position += y_velocity, 0), 0.2);
        if (x_position - 0.2 <= left.x() + wall_size || x_position + 0.2 >= right.x() - wall_size)
        {
            x_velocity *= -1;
        }
        if (y_position - 0.2 <= bottom.y() + wall_size || y_position + 0.2 >= top.y() - wall_size)
        {
            y_velocity *= -1;
        }

        //move balls
        *d_world = new hitable_list(d_list, 6);
    }
}
```

The collision is calculated by working out if the distance between the spheres position adjusted for the radius is intersecting with the wall position adjusted for the wall radius.

2. In order to achieve this:

I modified the Sphere class so that it contained a vec3 named colour to store its rgb values and modified the hit class so that hit_record had a colour variable which would be set when the ray hits the sphere as shown below

```
class sphere : public hitable {
public:
    __device__ sphere() {}
    __device__ sphere(vec3 cen, vec3 col, vec3 vec, float r) : center(cen), radius(r), colour(col), vector(vec) {};
    __device__ virtual bool hit(const ray& r, float tmin, float tmax, hit_record& rec) const;
    vec3 center;
    vec3 colour;
    vec3 vector;
    float radius;
};

__device__ bool sphere::hit(const ray& r, float t_min, float t_max, hit_record& rec) const {
    vec3 oc = r.origin() - center;
    float a = dot(r.direction(), r.direction());
    float b = dot(oc, r.direction());
    float c = dot(oc, oc) - radius * radius;
    float discriminant = b * b - a * c;
    if (discriminant > 0) {
        float temp = (-b - sqrt(discriminant)) / a;
        if (temp < t_max && temp > t_min) {
            rec.t = temp;
            rec.p = r.point_at_parameter(rec.t);
            rec.normal = (rec.p - center) / radius;
            rec.colour = colour;
            return true;
        }
        temp = (-b + sqrt(discriminant)) / a;
        if (temp < t_max && temp > t_min) {
            rec.t = temp;
            rec.p = r.point_at_parameter(rec.t);
            rec.normal = (rec.p - center) / radius;
            rec.colour = colour;
            return true;
        }
    }
    return false;
}
```

Modified the cast ray function so that when setting the colour, it applies the colour from the sphere that was hit to the shader.

```
__device__ vec3 castRay(const ray& r, hitable** world) {
    hit_record rec;
    if ((*world)->hit(r, 0.0, FLT_MAX, rec)) {
        vec3 result = 0.5f * vec3(rec.normal.x() + 1.0f, rec.normal.y() + 1.0f, rec.normal.z() + 1.0f);
        return result * (rec.colour/255);
    }
    else {
        vec3 unit_direction = unit_vector(r.direction());
        float t = 0.5f * (unit_direction.y() + 1.0f);
        return (1.0f - t) * vec3(1.0, 1.0, 1.0) + t * vec3(0.5, 0.7, 1.0);
    }
}
```

The results of this can be seen in sample output section 2

3. To achieve multiple spheres at the same time I modified the static device variable to store the following values

```
__device__ static int colour_index = 1;
__device__ static vec3 sphere_centres[3];
__device__ static vec3 sphere_velocitys[3];
__device__ static vec3 sphere_colours[3];
```

Then in the create world kernel if they are null then they are initialised and used to store the persistent sphere values the create world kernel now looks as follows

```

__global__ void create_world(hitable** d_list, hitable** d_world) {
    if (threadIdx.x == 0 && blockIdx.x == 0)
    {
        //define wall vec3s
        vec3 lef = vec3(-10002.0,      0,      0);
        vec3 rig = vec3( 10002.0,      0,      0);
        vec3 bot = vec3(      0, -10002.0,      0);
        vec3 top = vec3(      0,  10002.0,      0);
        vec3 bac = vec3(      0,      0, -10001.0);

        //define the colours
        vec3 blank = vec3(255, 255, 255);
        int number_of_colours = 6;
        vec3 colour_list[6];
        colour_list[0] = vec3(066, 245, 242);
        colour_list[1] = vec3(245, 066, 224);
        colour_list[2] = vec3(245, 242, 066);
        colour_list[3] = vec3(000, 255, 000);
        colour_list[4] = vec3(000, 000, 255);
        colour_list[5] = vec3(255, 000, 000);

        //initialise sphers statics
        if (sphere_velocities[0].x() == 0 && sphere_velocities[0].y() == 0 && sphere_velocities[0].z() == 0) { sphere_velocities[0] = vec3(0.01, 0.05, 0); sphere_colours[0] = vec3(150, 150, 150); }
        if (sphere_velocities[1].x() == 0 && sphere_velocities[1].y() == 0 && sphere_velocities[1].z() == 0) { sphere_velocities[1] = vec3(-0.03, -0.01, 0); sphere_colours[1] = vec3(150, 150, 150); }
        if (sphere_velocities[2].x() == 0 && sphere_velocities[2].y() == 0 && sphere_velocities[2].z() == 0) { sphere_velocities[2] = vec3(0.05, -0.02, 0); sphere_colours[2] = vec3(150, 150, 150); }
    }

    int wall_size = 10000;
    //Create the walls
    *(d_list + 0) = new sphere(lef, blank, vec3(0, 0, 0), wall_size);
    *(d_list + 1) = new sphere(rig, blank, vec3(0, 0, 0), wall_size);
    *(d_list + 2) = new sphere(bot, blank, vec3(0, 0, 0), wall_size);
    *(d_list + 3) = new sphere(top, blank, vec3(0, 0, 0), wall_size);
    *(d_list + 4) = new sphere(bac, blank, vec3(0, 0, 0), wall_size);

    //Modify balls
    for (int i = 0; i < 3; i++)
    {
        //move balls
        sphere_centres[i] += sphere_velocities[i];
        //Test Collisions
        if (sphere_centres[i].x() - 0.2 <= lef.x() + wall_size || sphere_centres[i].x() + 0.2 >= rig.x() - wall_size)
        {
            sphere_colours[i] = colour_list[colour_index];
            colour_index++;
            if (colour_index > 5)
            {
                colour_index = 0;
            }
            sphere_velocities[i] = vec3(sphere_velocities[i].x() * -1, sphere_velocities[i].y(), sphere_velocities[i].z());
        }
        if (sphere_centres[i].y() - 0.2 <= bot.y() + wall_size || sphere_centres[i].y() + 0.2 >= top.y() - wall_size)
        {
            sphere_colours[i] = colour_list[colour_index];
            colour_index++;
            if (colour_index > 5)
            {
                colour_index = 0;
            }
            sphere_velocities[i] = vec3(sphere_velocities[i].x(), sphere_velocities[i].y() * -1, sphere_velocities[i].z());
        }
        *(d_list + i + 5) = new sphere(sphere_centres[i], sphere_colours[i], sphere_velocities[i], 0.2);
    }
    *d_world = new hitable_list(d_list, 8);
}

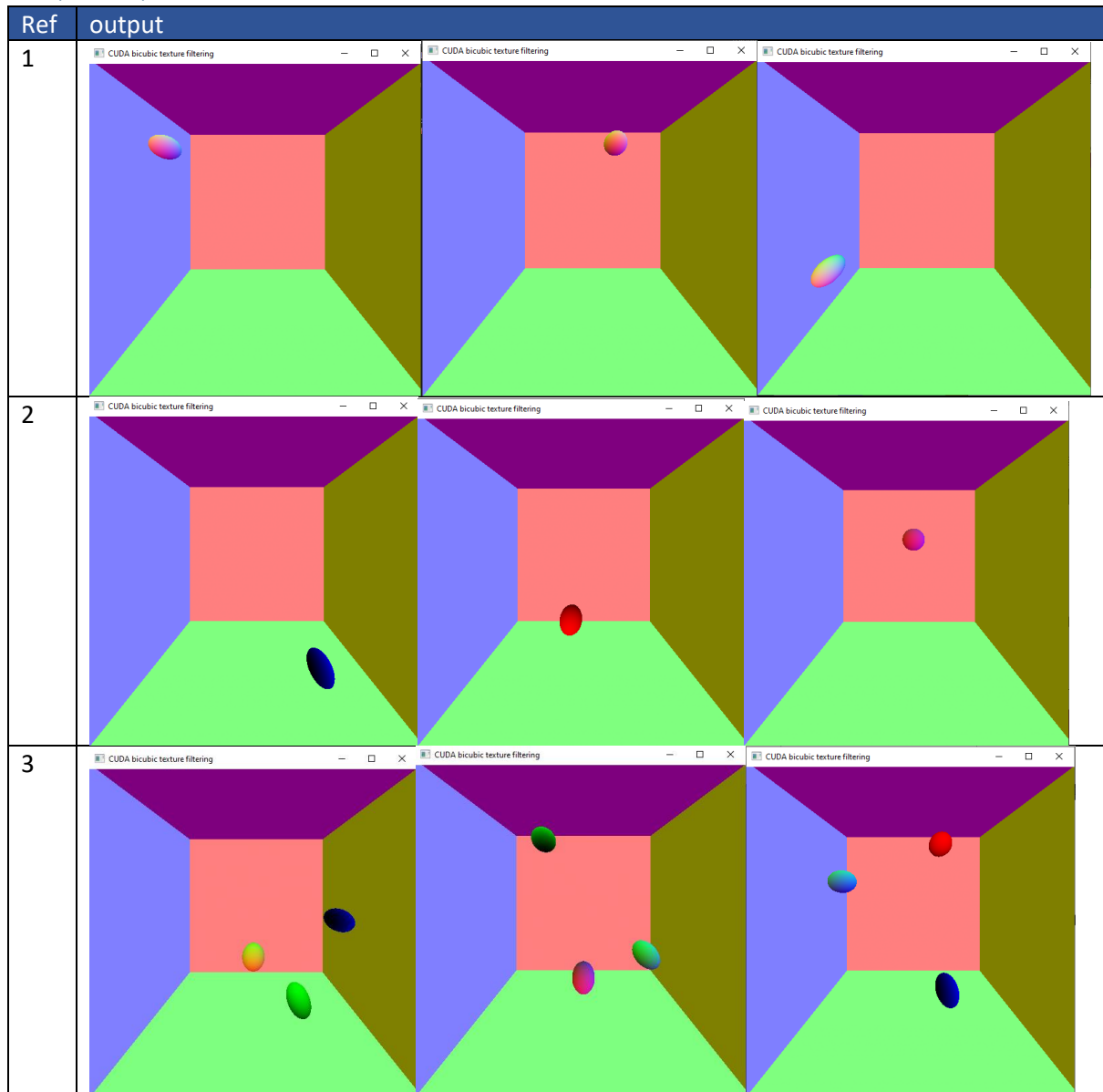
```

The above code will create 3 moving balls within the walls and the results can be seen in sample output ref 3.

Test data:

N/A

Sample output:



Reflection:

Getting the shading to apply to the coloured balls lead to some difficulty as I had not considered applying the colour to the normal value to achieve the shading results.

Metadata: