600086 Lab Book

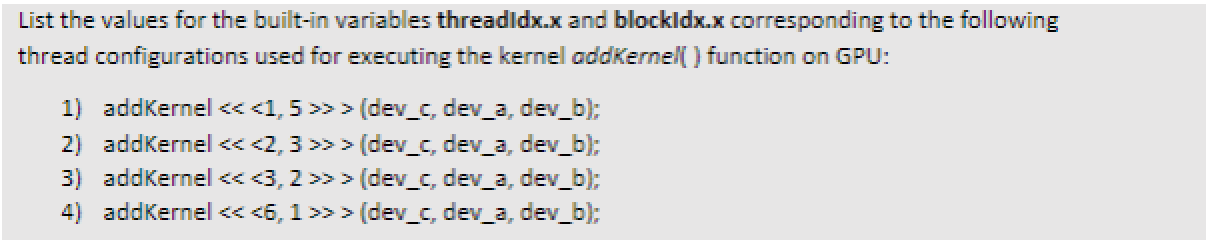
Week 2 – Lab B

Date: 17th Feb 2022

Callum Gray

Q1. Understand the block and thread indices

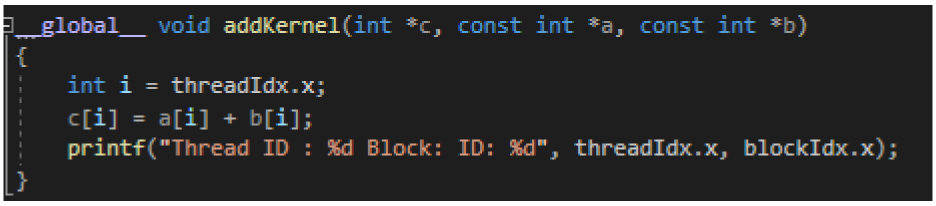
Question:



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:

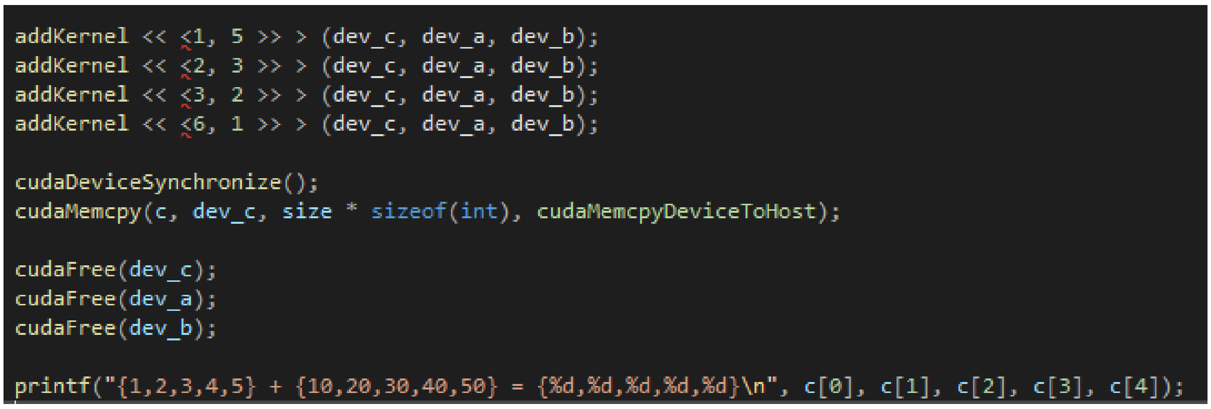
1) addKernel << <1, 5 >> > (dev_c, dev_a, dev_b);
2) addKernel << <2, 3 >> > (dev_c, dev_a, dev_b);
3) addKernel << <3, 2 >> > (dev_c, dev_a, dev_b);
4) addKernel << <6, 1 >> > (dev_c, dev_a, dev_b);

Solution:



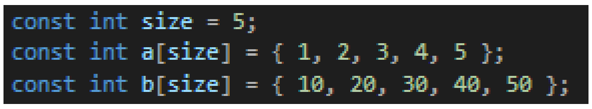
```
__global__ void addKernel(int *c, const int *a, const int *b)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
    printf("Thread ID : %d Block: ID: %d", threadIdx.x, blockIdx.x);
}
```

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



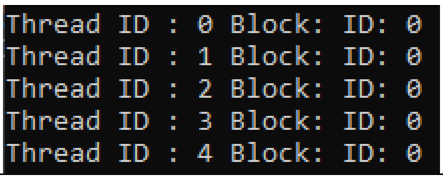
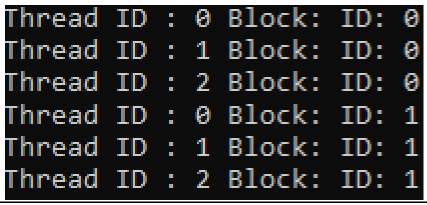
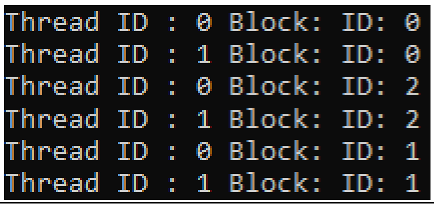
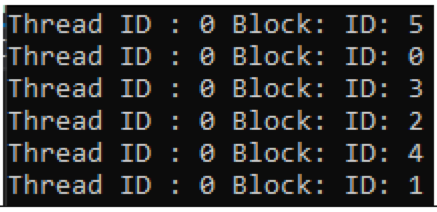
```
addKernel << <1, 5 >> > (dev_c, dev_a, dev_b);
addKernel << <2, 3 >> > (dev_c, dev_a, dev_b);
addKernel << <3, 2 >> > (dev_c, dev_a, dev_b);
addKernel << <6, 1 >> > (dev_c, dev_a, dev_b);

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:

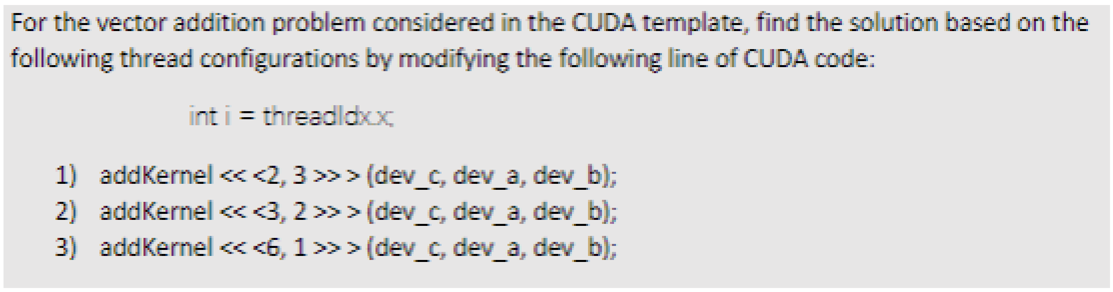
Threadidx.x, blockidx.x

Further information:

N/A

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

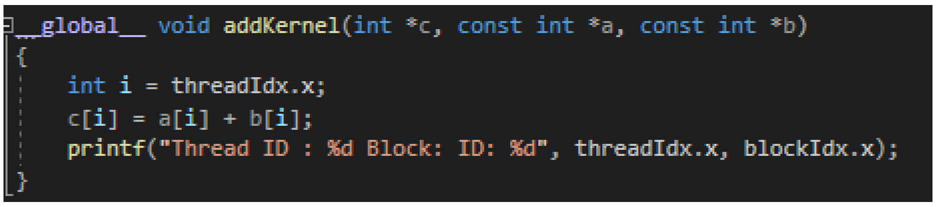
Question:


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:

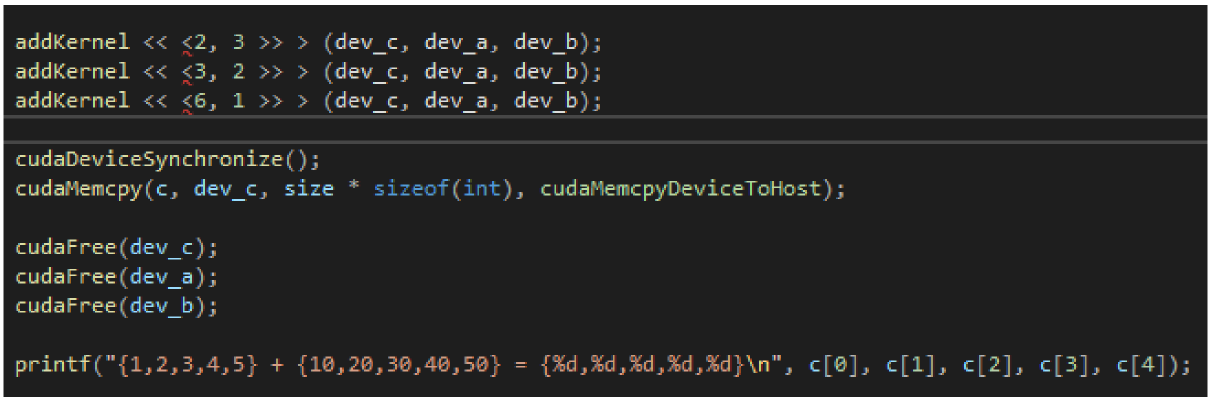
```
int i = threadIdx.x;
```

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);

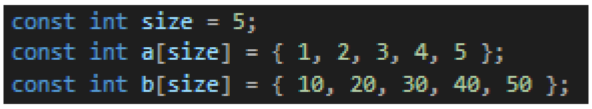
Solution:


```
__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);
}
```

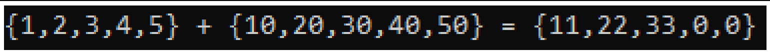
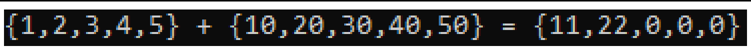
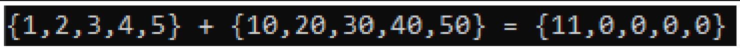
In the below image each add kernel function is run separately and the others commented out .


```
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:

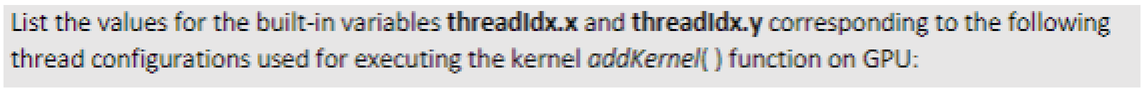
Multithread

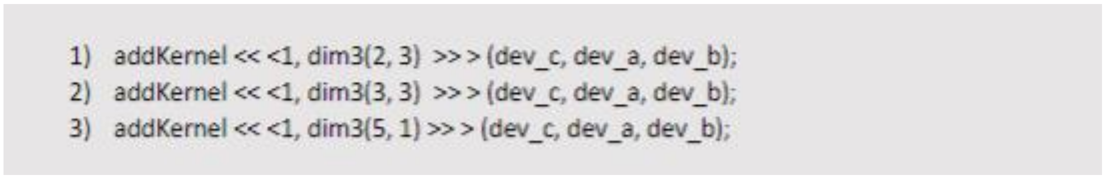
Further information:

N/A

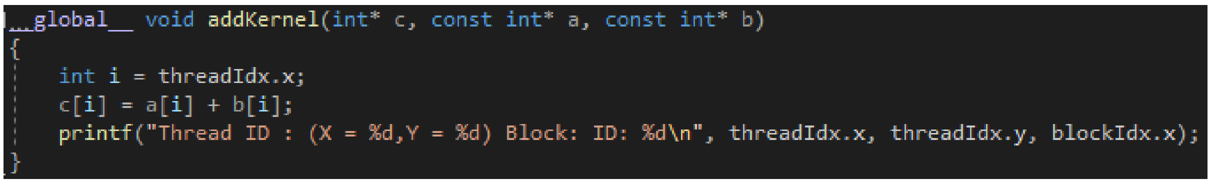
Q3. Exercise 3. Understand the thread indices for 2D blocks

Question:

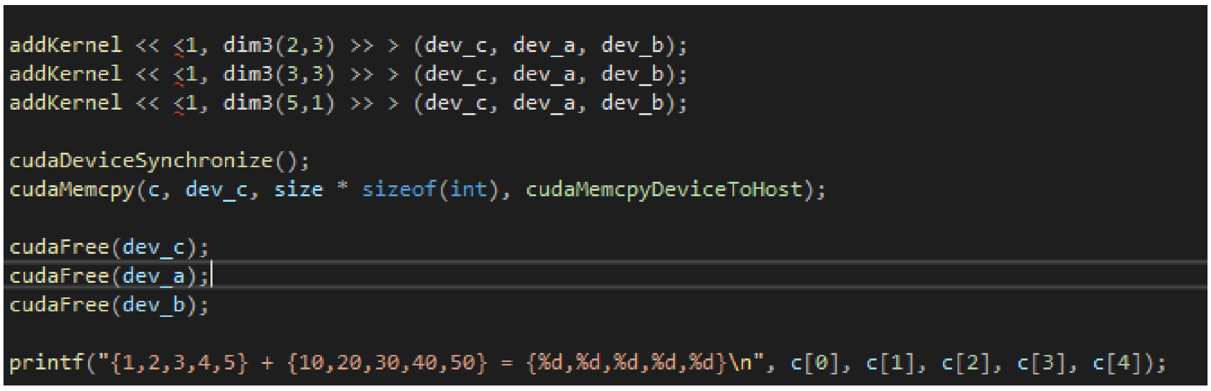

List the values for the built-in variables **threadIdx.x** and **threadIdx.y** corresponding to the following thread configurations used for executing the kernel *addKernel*( ) function on GPU:


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);

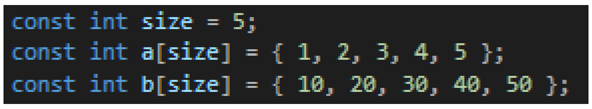
Solution:


```
__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 .

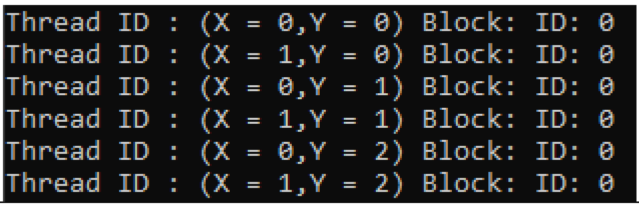

```
addKernel << <1, dim3(2,3) >> > (dev_c, dev_a, dev_b);
addKernel << <1, dim3(3,3) >> > (dev_c, dev_a, dev_b);
addKernel << <1, dim3(5,1) >> > (dev_c, dev_a, dev_b);

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]);
```

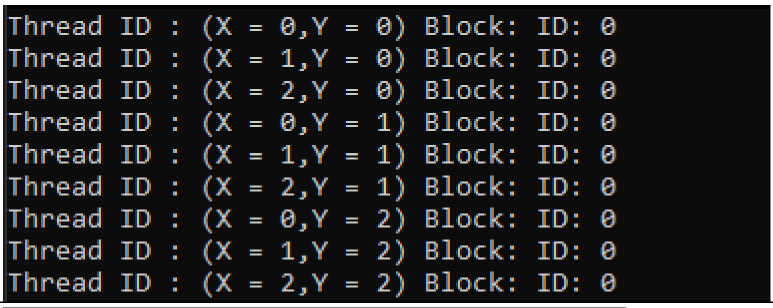
Test data:

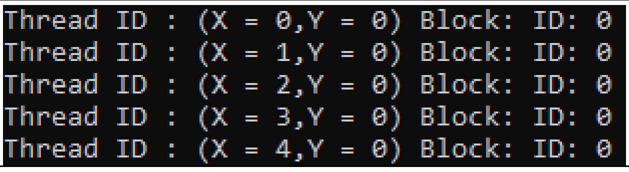

```
const int size = 5;
const int a[size] = { 1, 2, 3, 4, 5 };
const int b[size] = { 10, 20, 30, 40, 50 };
```

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:

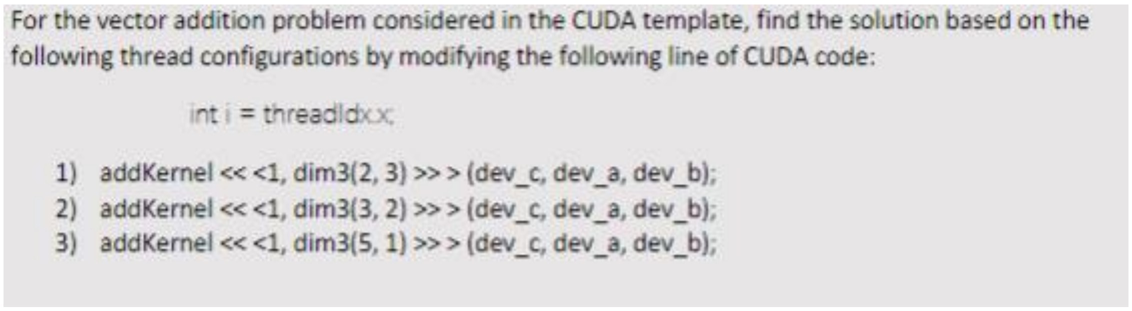
Multithread threadId.x threadId.y threadId.z

Further information:

N/A

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

Question:

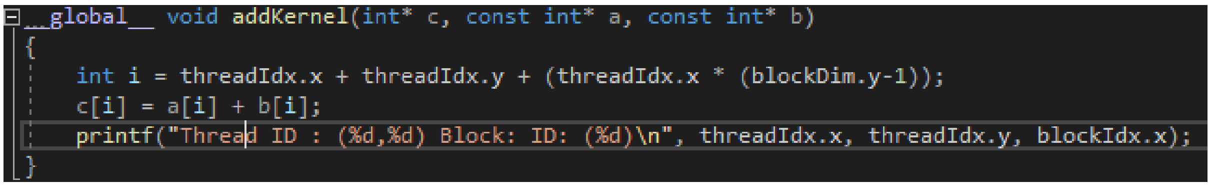


For the vector addition problem considered in the CUDA template, find the solution based on the following thread configurations by modifying the following line of CUDA code:

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

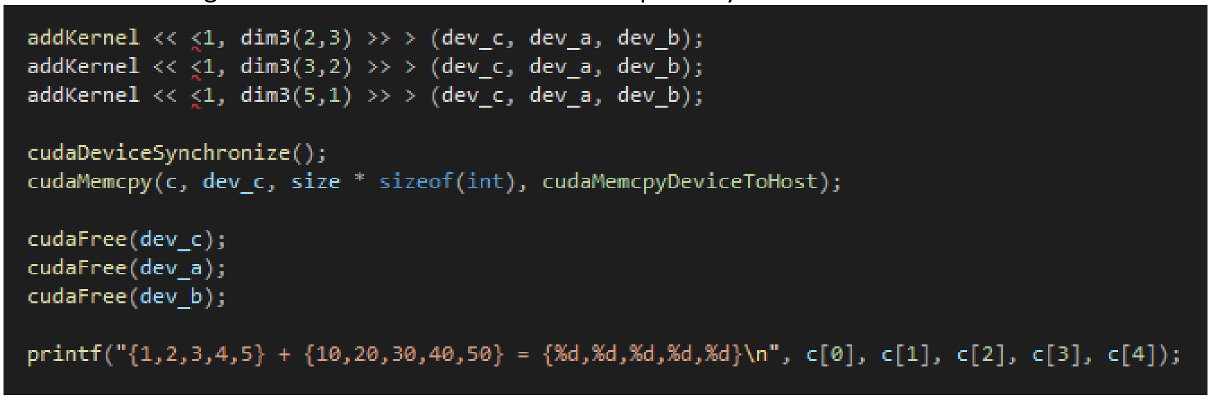
1) addKernel << <1, dim3(2, 3) >> > (dev_c, dev_a, dev_b);
2) addKernel << <1, dim3(3, 2) >> > (dev_c, dev_a, dev_b);
3) addKernel << <1, dim3(5, 1) >> > (dev_c, dev_a, dev_b);

Solution:



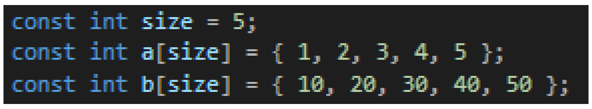
```
__global__ void addKernel(int* c, const int* a, const int* b)
{
    int i = threadIdx.x + threadIdx.y + (threadIdx.x * (blockDim.y-1));
    c[i] = a[i] + b[i];
    printf("Thread ID : (%d,%d) Block: ID: (%d)\n", threadIdx.x, threadIdx.y, blockIdx.x);
}
```

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]);
```

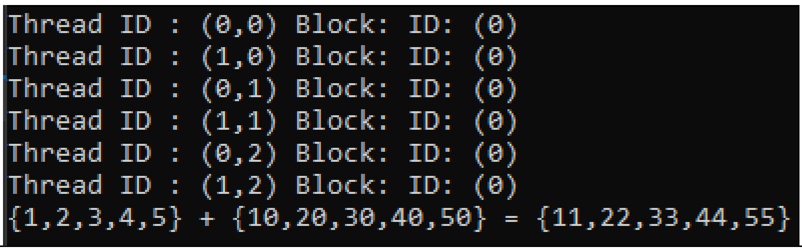
Test data:

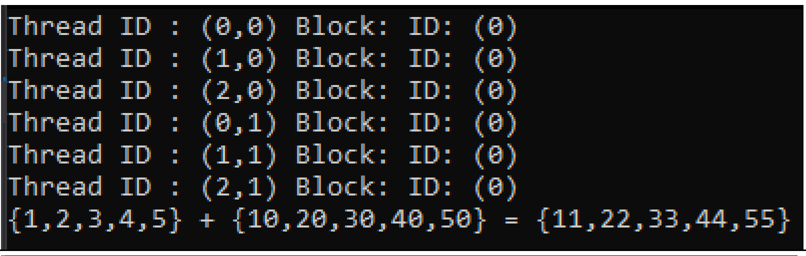


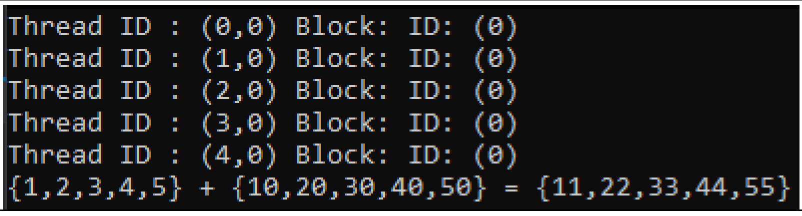
```
const int size = 5;
const int a[size] = { 1, 2, 3, 4, 5 };
const int b[size] = { 10, 20, 30, 40, 50 };
```

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}`

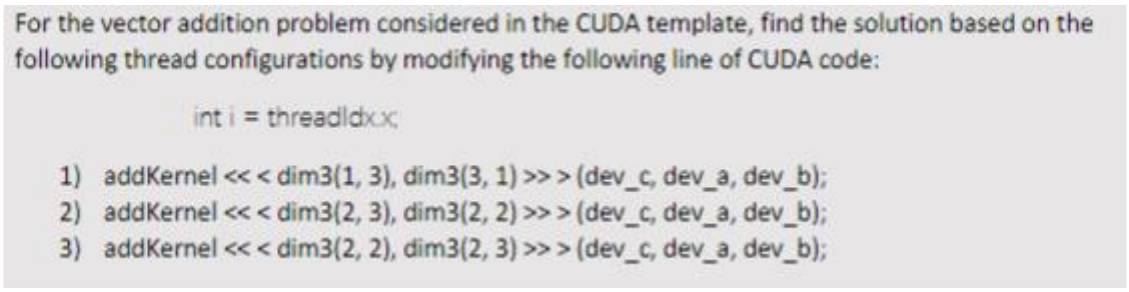
`addKernel << <1, dim3(5,1) >> > (dev_c, dev_a, dev_b);`	 `Thread ID : (0,0) Block: ID: (0)`
	`Thread ID : (1,0) Block: ID: (0)`
	`Thread ID : (2,0) Block: ID: (0)`
	`Thread ID : (3,0) Block: ID: (0)`
	`Thread ID : (4,0) Block: ID: (0)`
	`{1,2,3,4,5} + {10,20,30,40,50} = {11,22,33,44,55}`

Reflection:
The tasks' goal was to perform vector addition using a multi-dimensional array of threads however the vector addition only occure3d on the column number for the thread blocks.

Metadata:
Multithread threadId.x threadId.y threadId.z

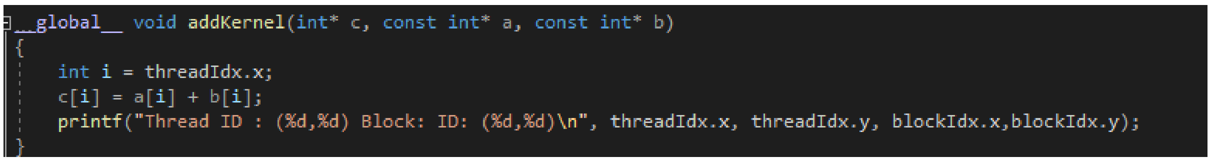
Further information:
Unsure how to resolve issue of only adding columns do we need to add an id checking method that will adapt to include rows as an overflow when there aren't enough columns?

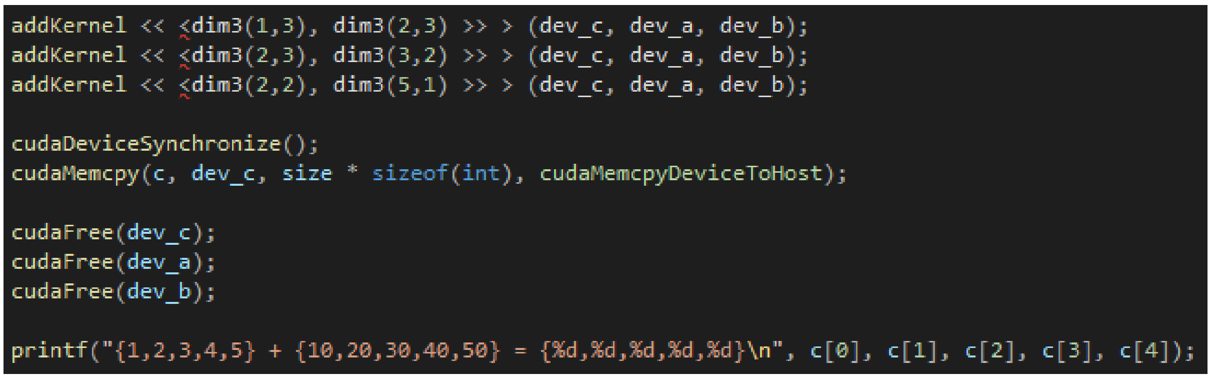
Q5. Exercise 5. Find vector addition using multiple 2D thread blocks

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;
```

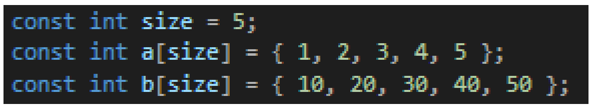
1) addKernel << < dim3(1, 3), dim3(3, 1) >> > (dev_c, dev_a, dev_b);
2) addKernel << < dim3(2, 3), dim3(2, 2) >> > (dev_c, dev_a, dev_b);
3) addKernel << < dim3(2, 2), dim3(2, 3) >> > (dev_c, dev_a, dev_b);

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?

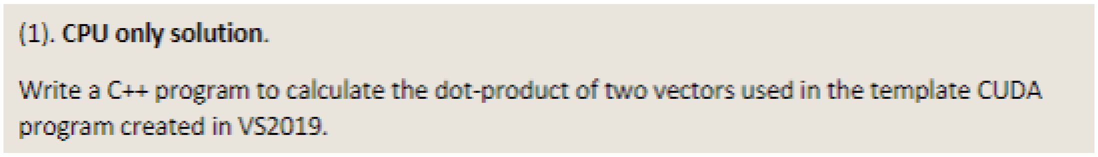
600086 Lab Book

Week 3 – Lab C

Date: 22nd Feb 2022

E1. Vector dot-product

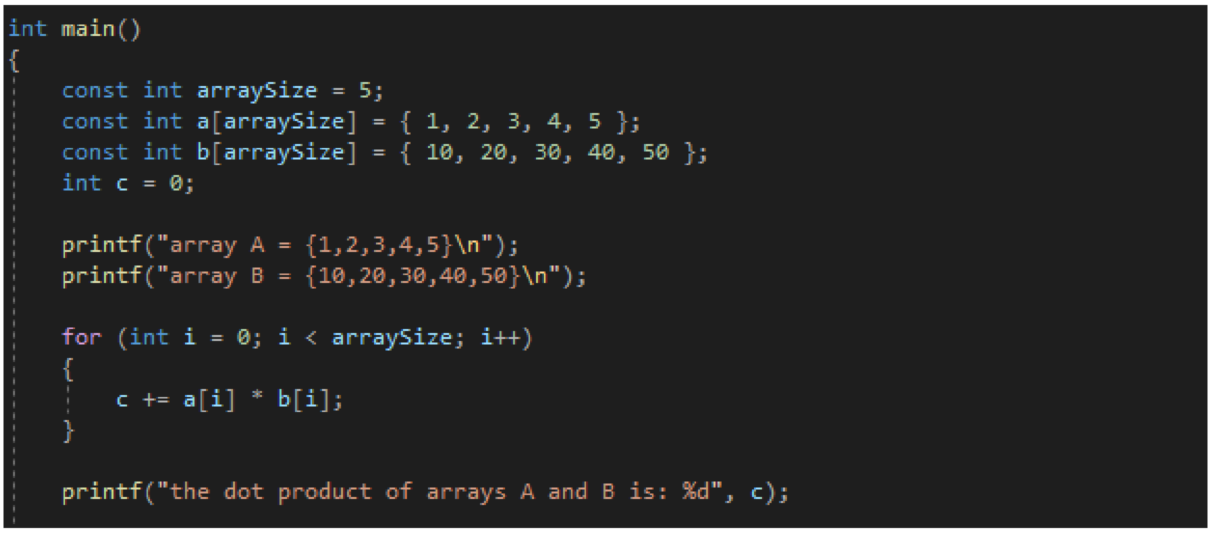
Question:



(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:

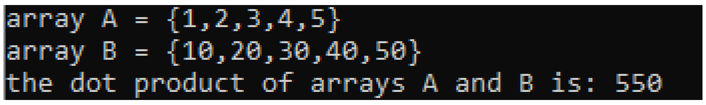


```
int main()
{
    const int arraySize = 5;
    const int a[arraySize] = { 1, 2, 3, 4, 5 };
    const int b[arraySize] = { 10, 20, 30, 40, 50 };
    int c = 0;

    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 **a**=(a_0, a_1, a_2, ..., a_{n-1}) and **b**=(b_0, b_1, b_2, ..., b_{n-1}), can be found in two steps:

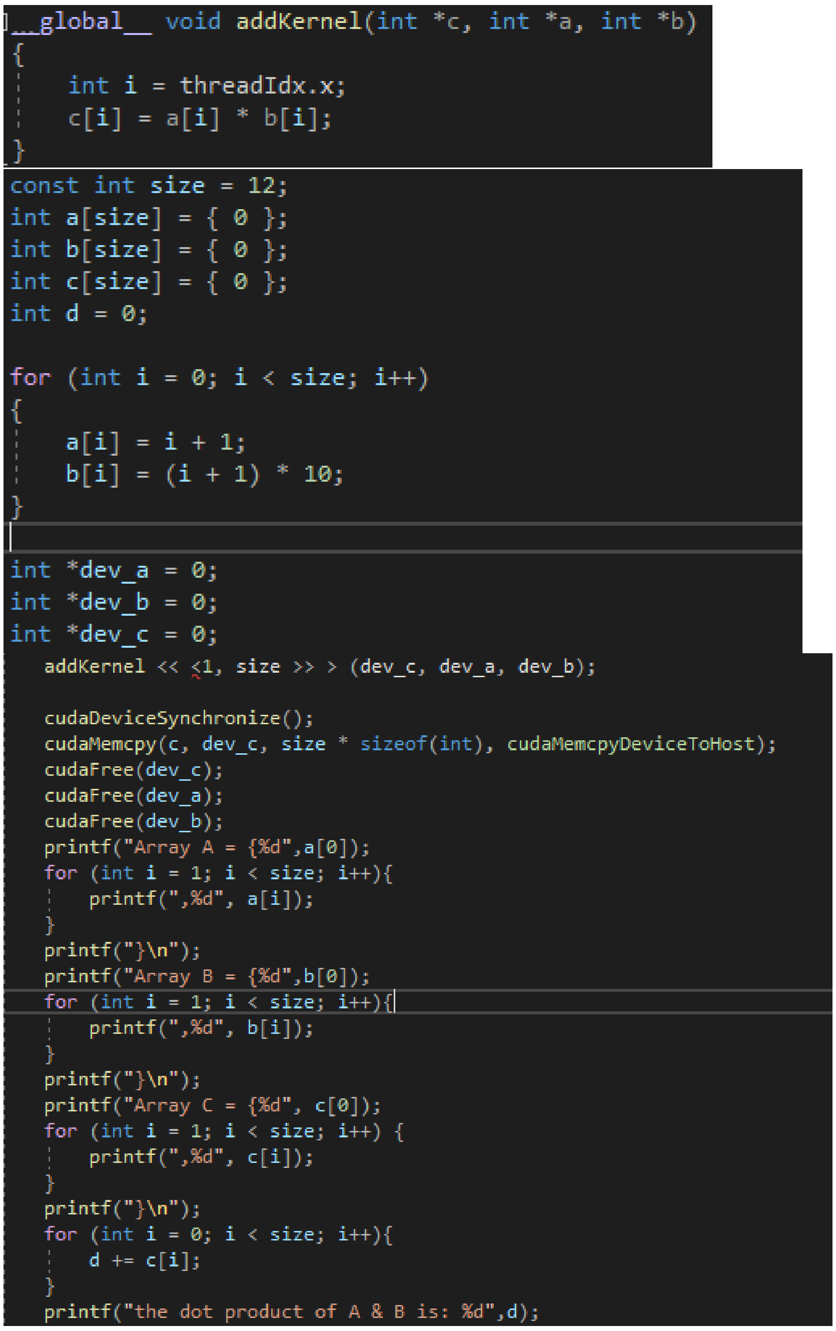
Step 1. Per-element multiplication: In this step, we calculate a vector **c**:

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

This task can be done in parallel on GPU.

Step 2. Calculate on CPU the sum of elements of vector **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:

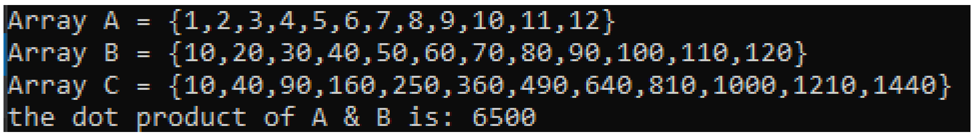


```
__global__ void addKernel(int *c, int *a, int *b)
{
    int i = threadIdx.x;
    c[i] = a[i] * b[i];
}
const int size = 12;
int a[size] = { 0 };
int b[size] = { 0 };
int c[size] = { 0 };
int d = 0;

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”

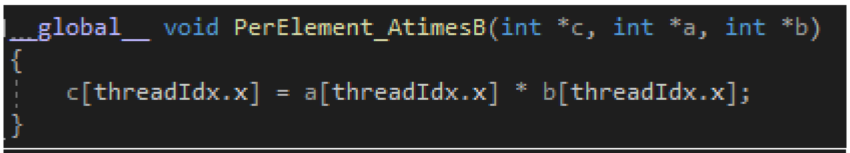
Further information:

E2. Vector dot-product using unified memory

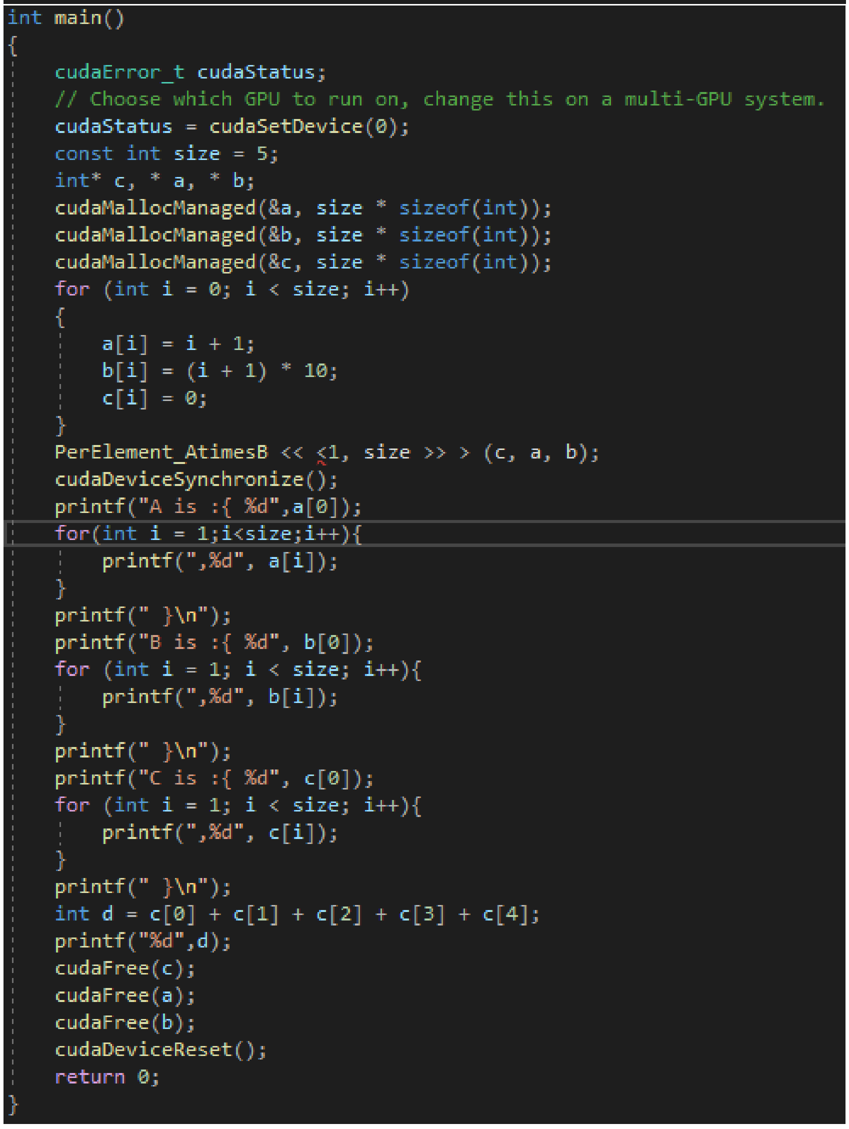
Question:

Exercise 2.1 Vector dot-product using managed memory

Solution:

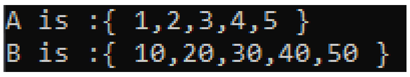


```
__global__ void PerElement_AtimesB(int *c, int *a, int *b)
{
    c[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];
}
```



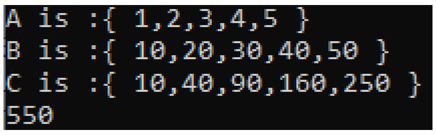
```
int main()
{
    cudaError_t cudaStatus;
    // Choose which GPU to run on, change this on a multi-GPU system.
    cudaStatus = cudaSetDevice(0);
    const int size = 5;
    int* c, * a, * b;
    cudaMallocManaged(&a, size * sizeof(int));
    cudaMallocManaged(&b, size * sizeof(int));
    cudaMallocManaged(&c, 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:

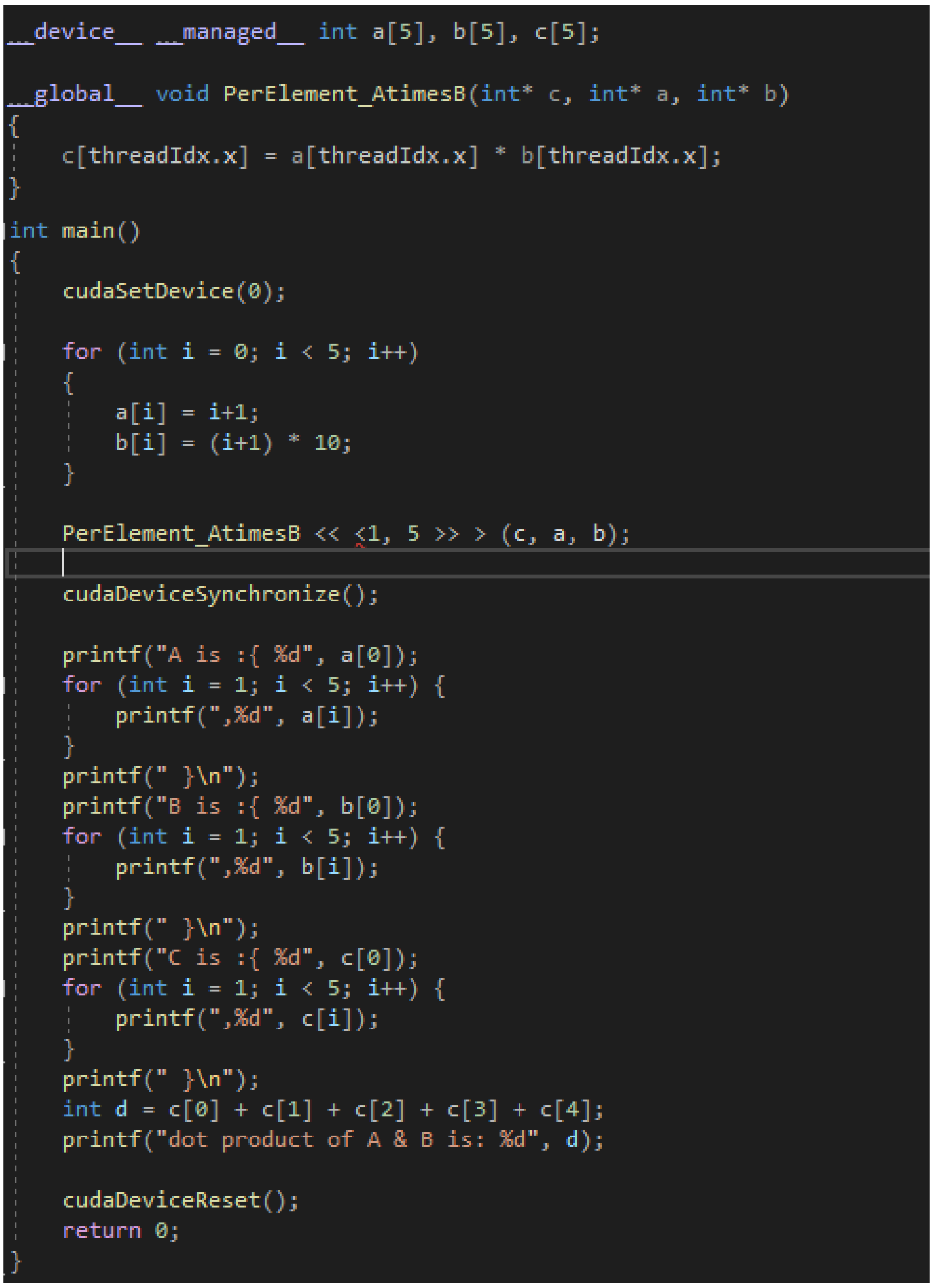


```
A is :{ 1,2,3,4,5 }
B is :{ 10,20,30,40,50 }
C is :{ 10,40,90,160,250 }
550
```

Question:

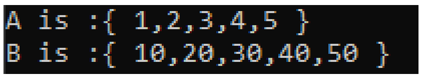
Exercise 2.3 Vector dot-product using GPU-declared __managed__ memory

Solution:



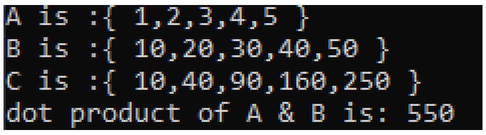
```
__device__ __managed__ int a[5], b[5], c[5];

__global__ void PerElement_AtimesB(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 cudaMallocManaaged() 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 sin 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”

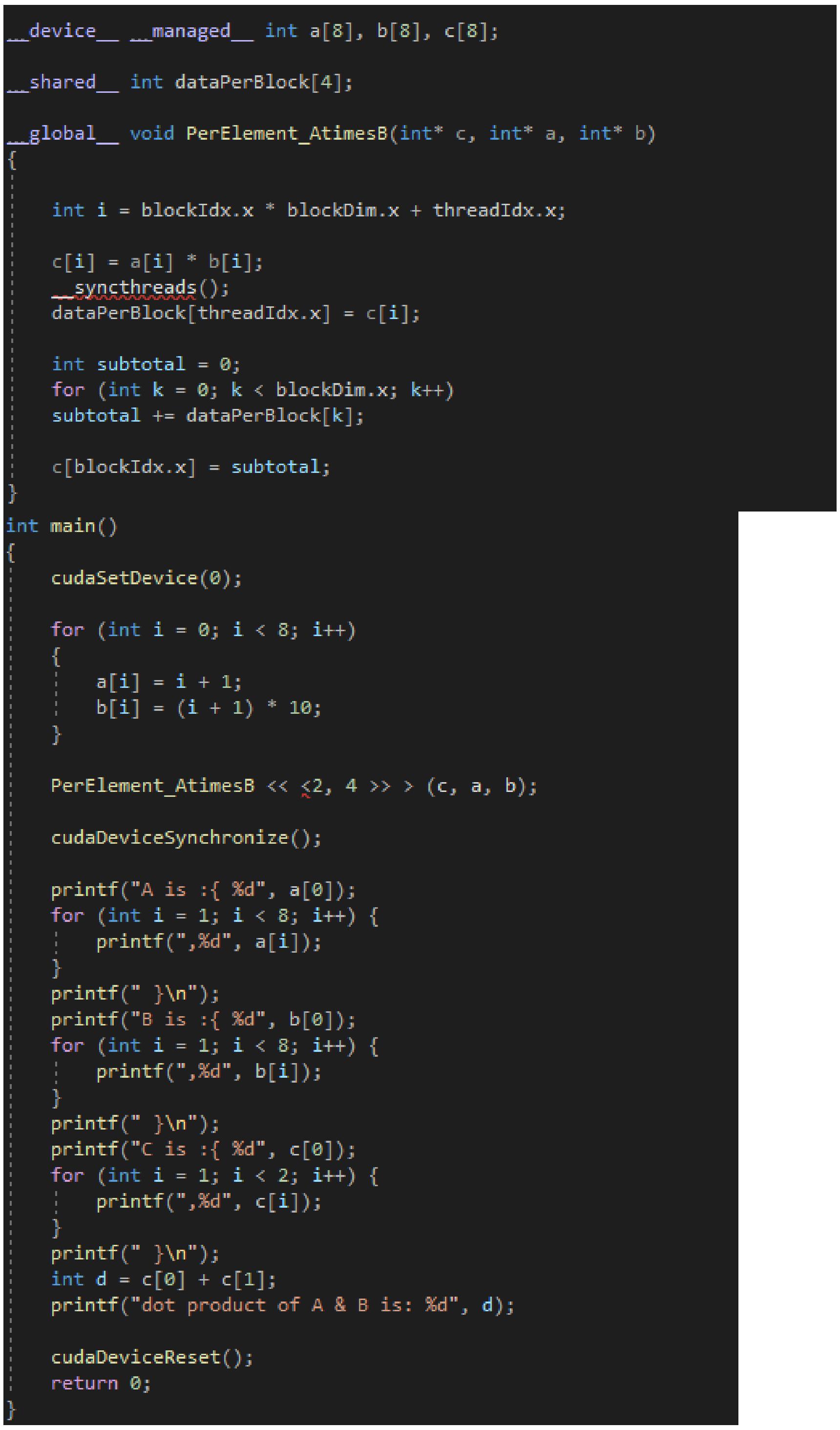
Further information:

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:

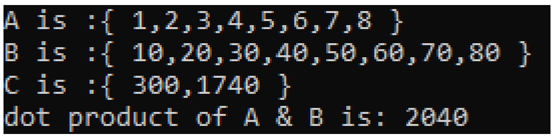


```
__device__ __managed__ int a[8], b[8], c[8];

__shared__ int dataPerBlock[4];

__global__ void PerElement_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];

    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
```

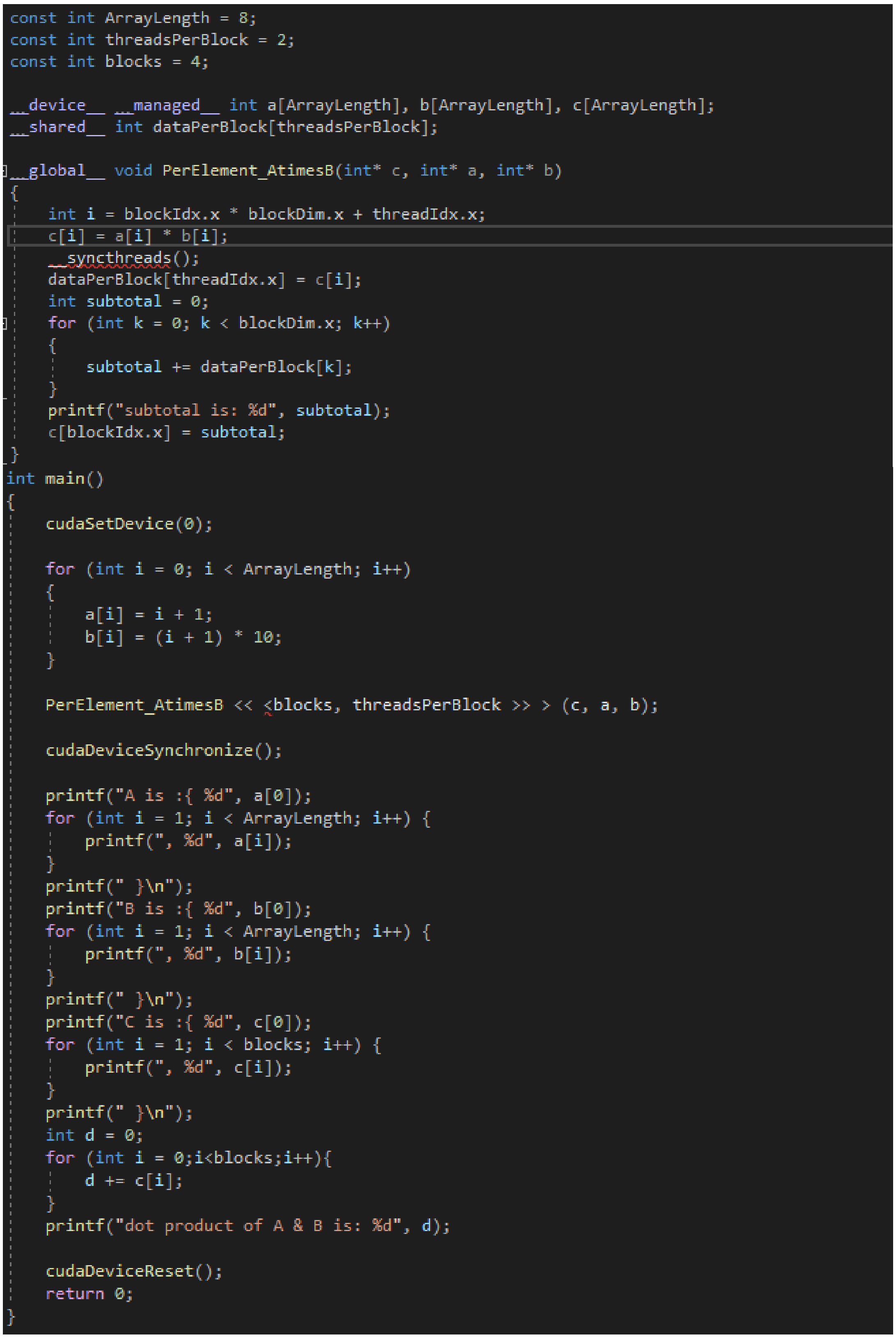
Reflection:

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 blocks*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

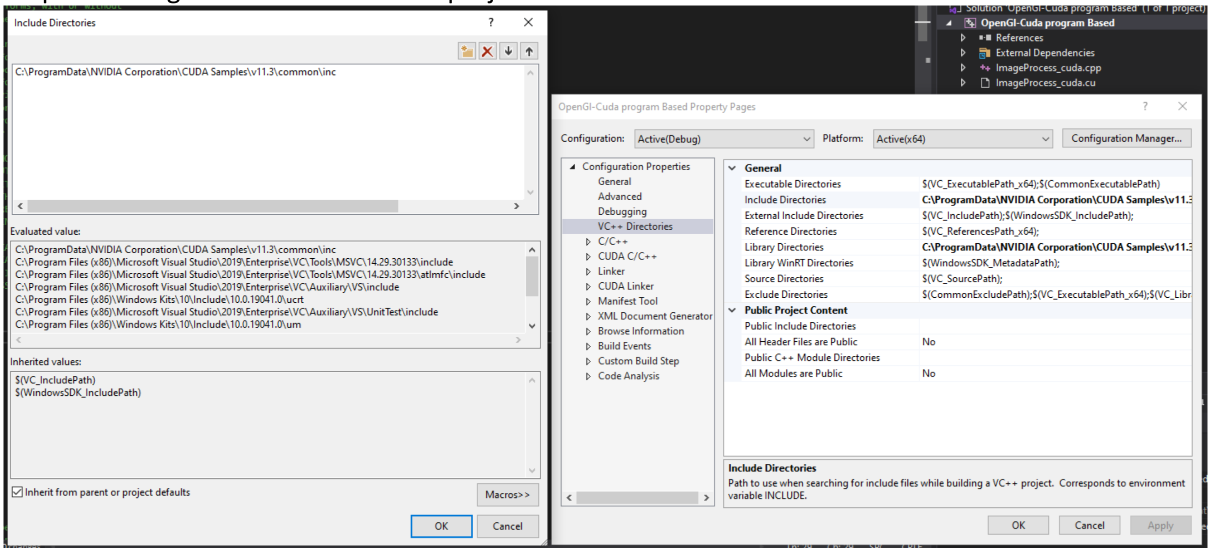
Question:

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

Solution:

No sample code to show

Step 1 : adding include directories to project



Include Directories

C:\ProgramData\NVIDIA Corporation\CUDA Samples\v11.3\common\inc

Evaluated value:

```
C:\ProgramData\NVIDIA Corporation\CUDA Samples\v11.3\common\inc
C:\Program Files (x86)\Microsoft Visual Studio\2019\Enterprise\VC\Tools\MSVC\14.29.30133\include
C:\Program Files (x86)\Microsoft Visual Studio\2019\Enterprise\VC\Tools\MSVC\14.29.30133\atlmfc\include
C:\Program Files (x86)\Microsoft Visual Studio\2019\Enterprise\VC\Auxiliary\VS\include
C:\Program Files (x86)\Windows Kits\10\Include\10.0.19041.0\ucrt
C:\Program Files (x86)\Microsoft Visual Studio\2019\Enterprise\VC\Auxiliary\VS\UnitTest\include
C:\Program Files (x86)\Windows Kits\10\Include\10.0.19041.0\um
```

Inherited values:

```
$(VC_IncludePath)
$(WindowsSDK_IncludePath)
```

☑ Inherit from parent or project defaults

Solution 'OpenGl-Cuda program Based' (1 of 1 project)
OpenGl-Cuda program Based
References
External Dependencies
ImageProcess_cuda.cpp
ImageProcess_cuda.cu

OpenGl-Cuda program Based Property Pages

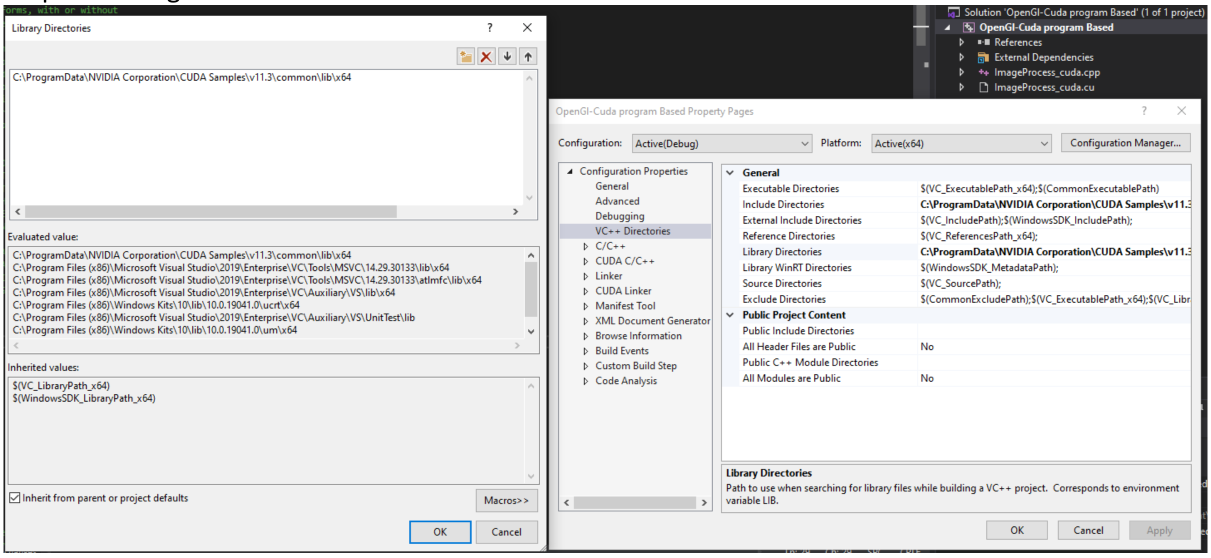
Configuration: Active(Debug) Platform: Active(x64)

Configuration Properties > VC++ Directories

General	
Executable Directories	$(VC_ExecutablePath_x64);$(CommonExecutablePath)
Include Directories	**C:\ProgramData\NVIDIA Corporation\CUDA Samples\v11.3**
External Include Directories	$(VC_IncludePath);$(WindowsSDK_IncludePath);
Reference Directories	$(VC_ReferencesPath_x64);
Library Directories	**C:\ProgramData\NVIDIA Corporation\CUDA Samples\v11.3**
Library WinRT Directories	$(WindowsSDK_MetadataPath);
Source Directories	$(VC_SourcePath);
Exclude Directories	$(CommonExcludePath);$(VC_ExecutablePath_x64);$(VC_Libr
Public Project Content	
Public Include Directories	
All Header Files are Public	No
Public C++ Module Directories	
All Modules are Public	No

Include Directories
Path to use when searching for include files while building a VC++ project. Corresponds to environment variable INCLUDE.

Step 2 : adding Lib directories



Library Directories

C:\ProgramData\NVIDIA Corporation\CUDA Samples\v11.3\common\lib\x64

Evaluated value:

```
C:\ProgramData\NVIDIA Corporation\CUDA Samples\v11.3\common\lib\x64
C:\Program Files (x86)\Microsoft Visual Studio\2019\Enterprise\VC\Tools\MSVC\14.29.30133\lib\x64
C:\Program Files (x86)\Microsoft Visual Studio\2019\Enterprise\VC\Tools\MSVC\14.29.30133\atlmfc\lib\x64
C:\Program Files (x86)\Microsoft Visual Studio\2019\Enterprise\VC\Auxiliary\VS\lib\x64
C:\Program Files (x86)\Windows Kits\10\lib\10.0.19041.0\ucrt\x64
C:\Program Files (x86)\Microsoft Visual Studio\2019\Enterprise\VC\Auxiliary\VS\UnitTest\lib
C:\Program Files (x86)\Windows Kits\10\lib\10.0.19041.0\um\x64
```

Inherited values:

```
$(VC_LibraryPath_x64)
$(WindowsSDK_LibraryPath_x64)
```

☑ Inherit from parent or project defaults

Solution 'OpenGl-Cuda program Based' (1 of 1 project)
OpenGl-Cuda program Based
References
External Dependencies
ImageProcess_cuda.cpp
ImageProcess_cuda.cu

OpenGl-Cuda program Based Property Pages

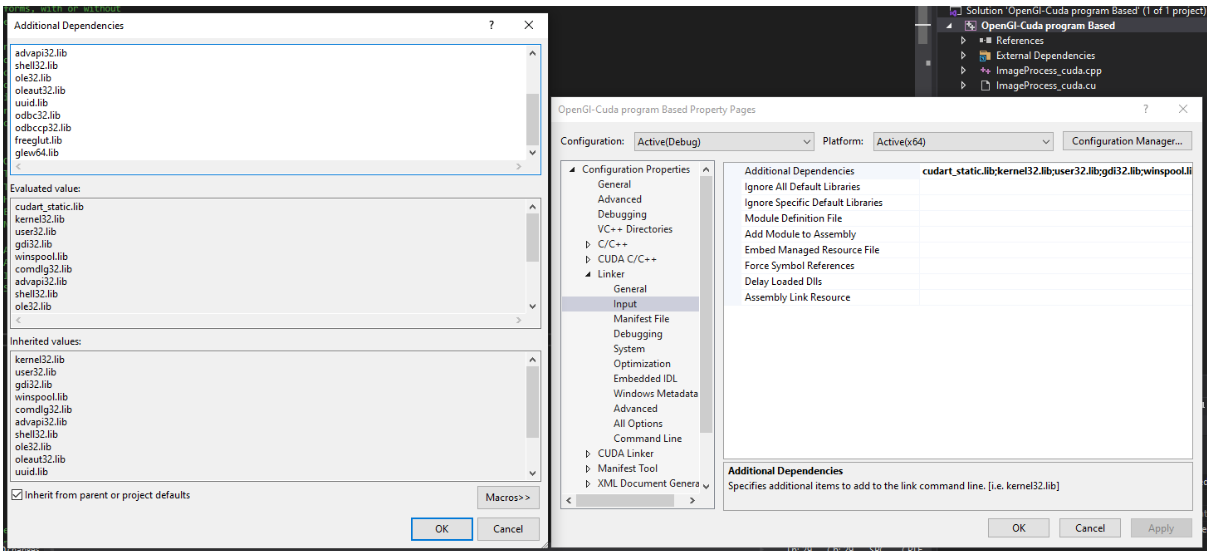
Configuration: Active(Debug) Platform: Active(x64)

Configuration Properties > VC++ Directories

General	
Executable Directories	$(VC_ExecutablePath_x64);$(CommonExecutablePath)
Include Directories	**C:\ProgramData\NVIDIA Corporation\CUDA Samples\v11.3**
External Include Directories	$(VC_IncludePath);$(WindowsSDK_IncludePath);
Reference Directories	$(VC_ReferencesPath_x64);
Library Directories	**C:\ProgramData\NVIDIA Corporation\CUDA Samples\v11.3**
Library WinRT Directories	$(WindowsSDK_MetadataPath);
Source Directories	$(VC_SourcePath);
Exclude Directories	$(CommonExcludePath);$(VC_ExecutablePath_x64);$(VC_Libr
Public Project Content	
Public Include Directories	
All Header Files are Public	No
Public C++ Module Directories	
All Modules are Public	No

Library Directories
Path to use when searching for library files while building a VC++ project. Corresponds to environment variable LIB.

Step 3 : adding to the Linker files


Additional Dependencies

advapi32.lib
shell32.lib
ole32.lib
oleaut32.lib
uuid.lib
odbc32.lib
odbccp32.lib
freeglut.lib
glew64.lib

Evaluated value:

cudart_static.lib
kernel32.lib
user32.lib
gdi32.lib
winspool.lib
comdlg32.lib
advapi32.lib
shell32.lib
ole32.lib

Inherited values:

kernel32.lib
user32.lib
gdi32.lib
winspool.lib
comdlg32.lib
advapi32.lib
shell32.lib
ole32.lib
oleaut32.lib
uuid.lib

☑ Inherit from parent or project defaults

OpenGl-Cuda program Based Property Pages

Configuration: Active(Debug) Platform: Active(x64)

Additional Dependencies	cudart_static.lib;kernel32.lib;user32.lib;gdi32.lib;winspool.li
Ignore All Default Libraries	
Ignore Specific Default Libraries	
Module Definition File	
Add Module to Assembly	
Embed Managed Resource File	
Force Symbol References	
Delay Loaded Dlls	
Assembly Link Resource	

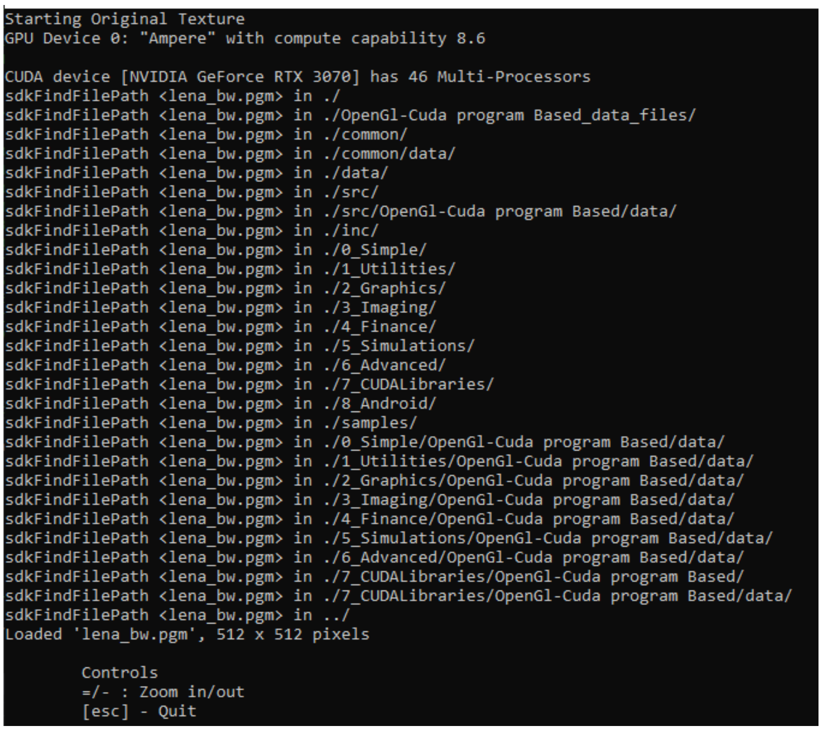
Additional Dependencies
Specifies additional items to add to the link command line. [i.e. kernel32.lib]

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);
```

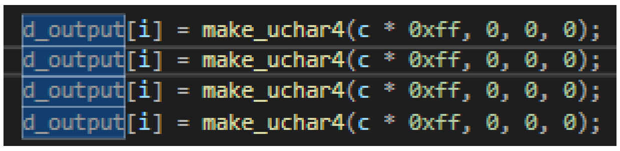
and then

```
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].

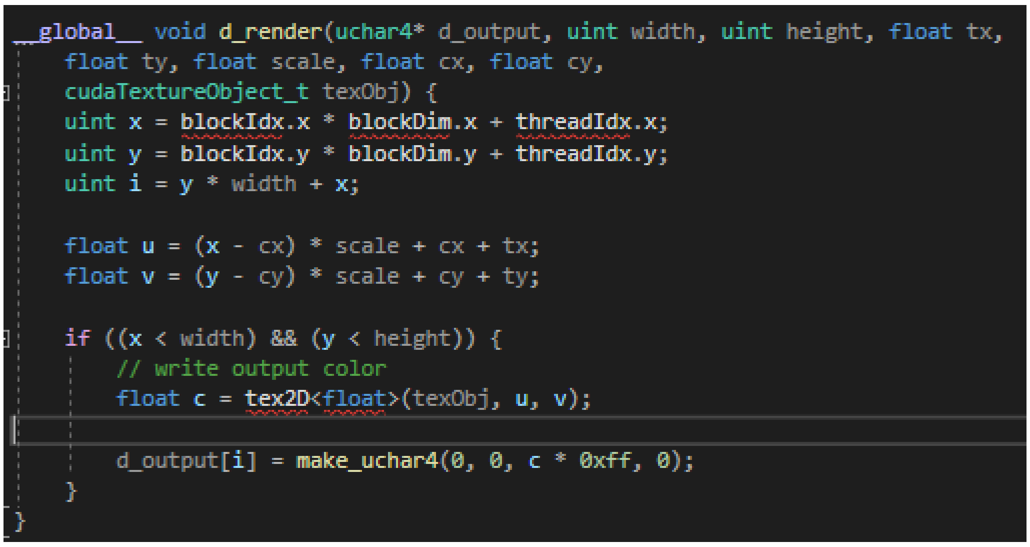
c) Now modify the value d_output[i] using image pixel value c read from image location at (u, v) with the following colour and observe how the image colour is changed.
d_output[i] = make_uchar4(0, 0, c*0xff, 0);

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

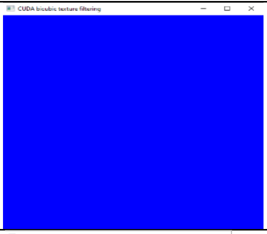
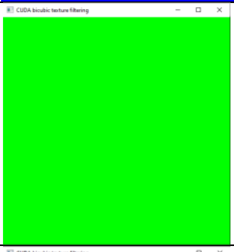
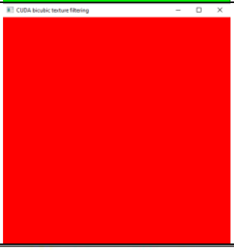
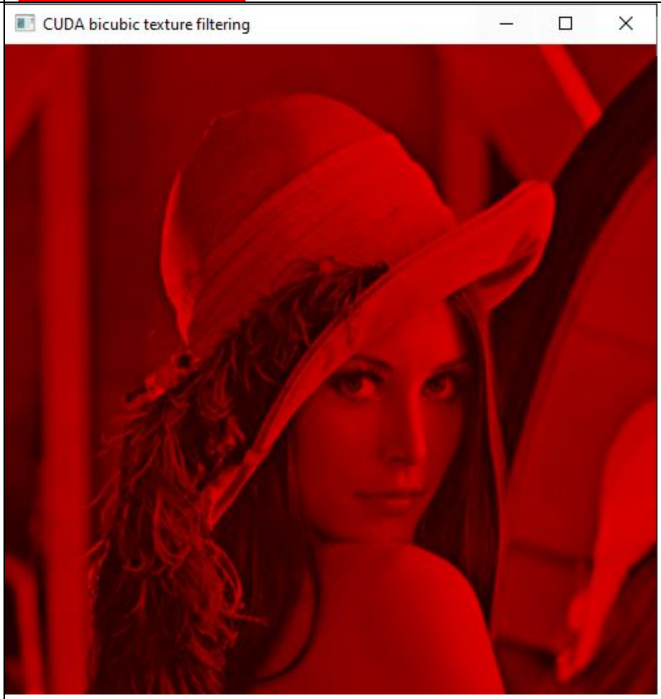


```
__global__ void d_render(uchar4* d_output, uint width, uint height, float tx,
    float ty, float scale, float cx, float cy,
    cudaTextureObject_t texObj) {
    uint x = blockIdx.x * blockDim.x + threadIdx.x;
    uint y = blockIdx.y * blockDim.y + threadIdx.y;
    uint i = y * width + x;

    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);	 CUDA bicubic texture filtering
d_output[i] = make_uchar4(0, 0xff, 0, 0);	 CUDA bicubic texture filtering
d_output[i] = make_uchar4(0, 0, 0xff, 0);	 CUDA bicubic texture filtering
d_output[i] = make_uchar4(0, 0, 0xff, 0);	 CUDA bicubic texture filtering

Reflection:

Nothing to report was fairly perfunctory

Exercise 3. Image Transformation

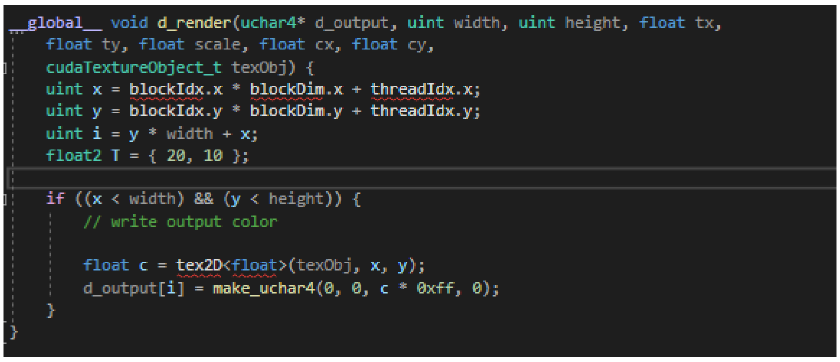
Question:

Demonstrate Image transformation.

Solution:

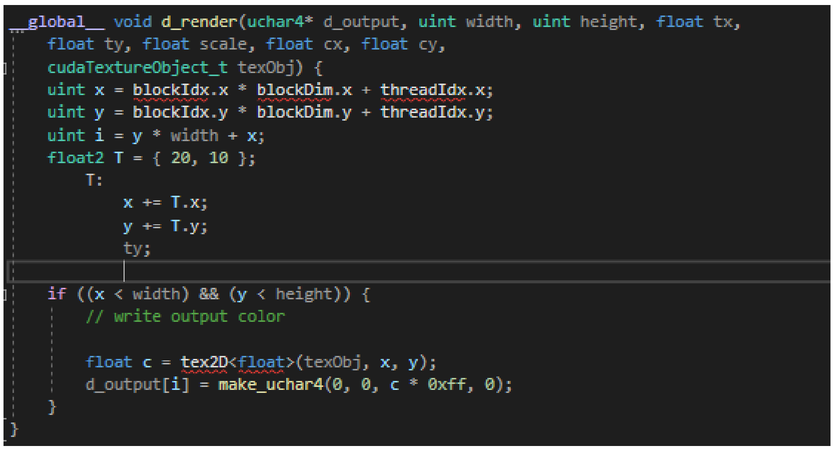
Translate the image.

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



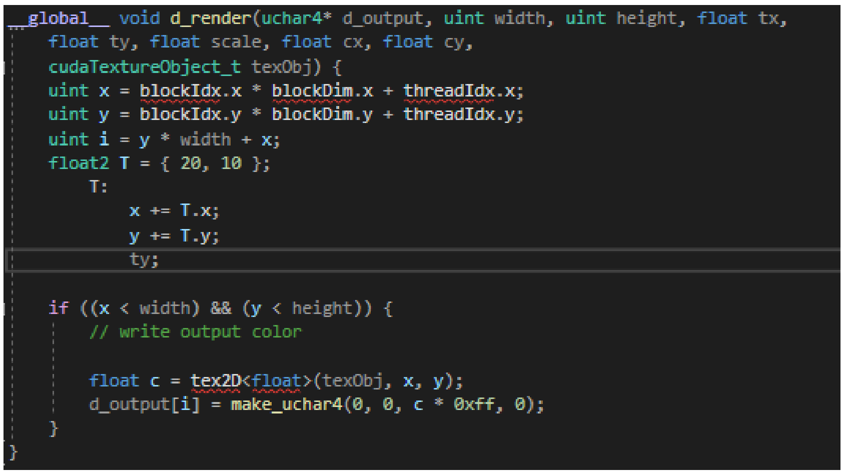
```
__global__ void d_render(uchar4* d_output, uint width, uint height, float tx,
    float ty, float scale, float cx, float cy,
    cudaTextureObject_t texObj) {
    uint x = blockIdx.x * blockDim.x + threadIdx.x;
    uint y = blockIdx.y * blockDim.y + threadIdx.y;
    uint i = y * width + x;
    float2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

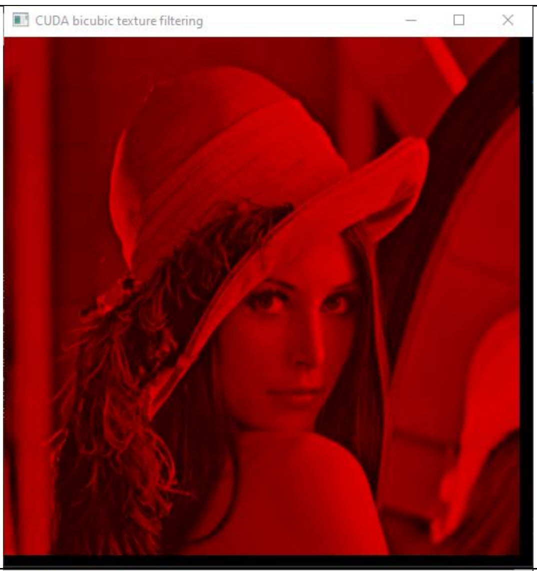
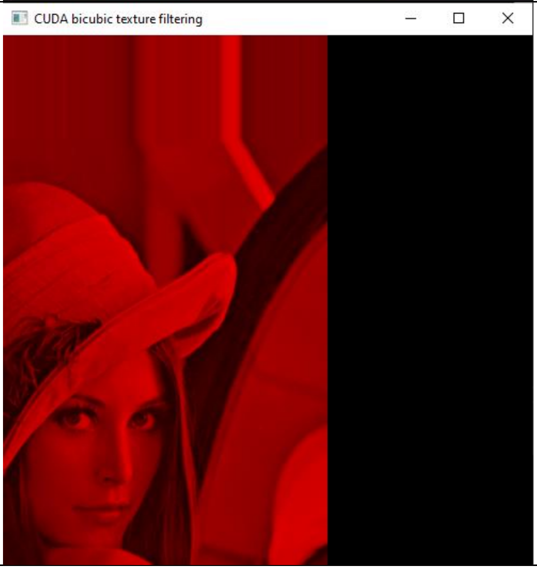
c. Read pixel colour with translated coordinates x, y: float c = tex2D(texObj, x, y);



```
__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?

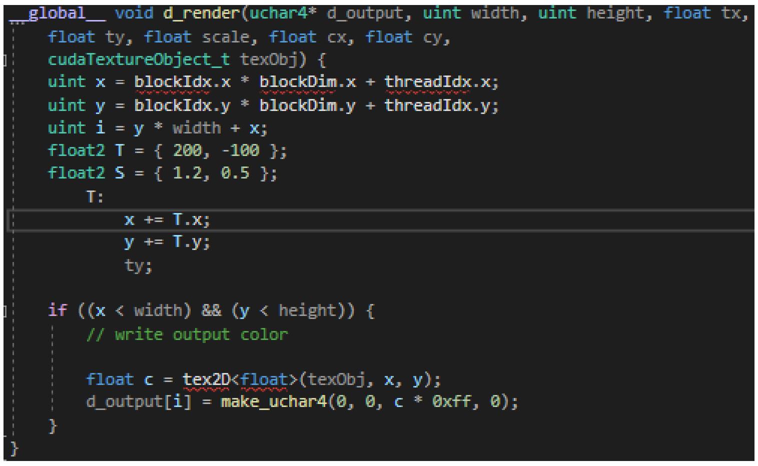
Question:

Demonstrate Image Scaling

Solution:

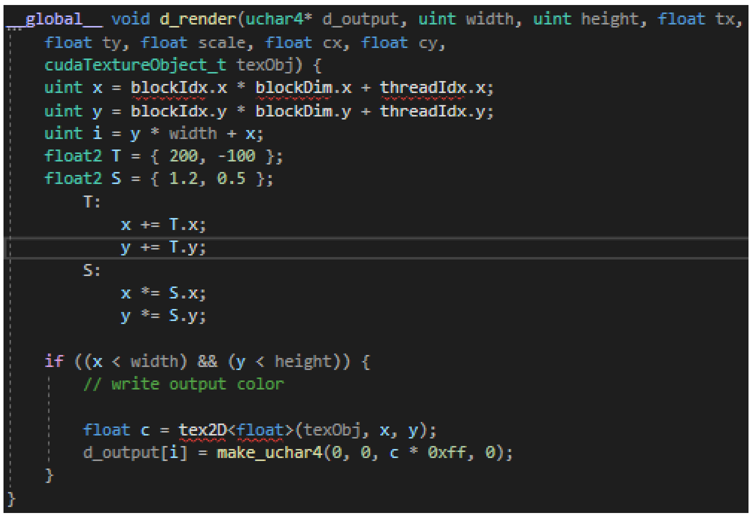
Scale the image

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



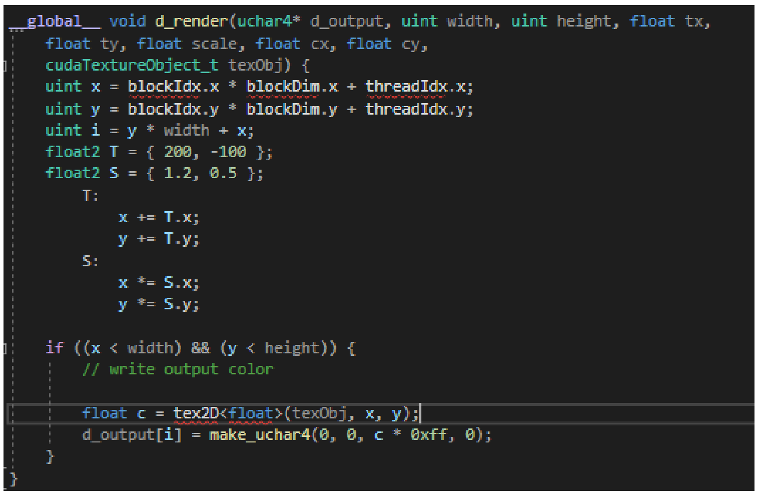
```
__global__ void d_render(uchar4* d_output, uint width, uint height, float tx,
    float ty, float scale, float cx, float cy,
    cudaTextureObject_t texObj) {
    uint x = blockIdx.x * blockDim.x + threadIdx.x;
    uint y = blockIdx.y * blockDim.y + threadIdx.y;
    uint i = y * width + x;
    float2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

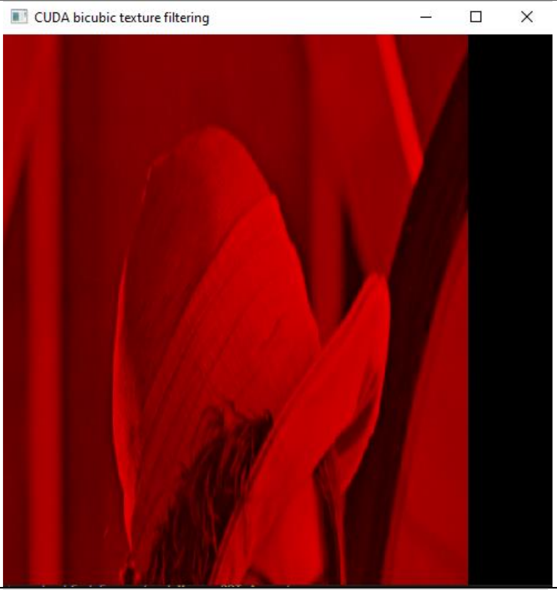
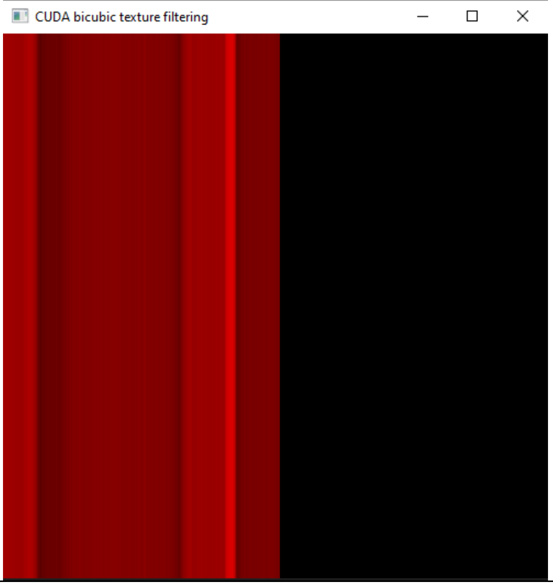
c. Read pixel colour with scaled coordinates x, y: float c = tex2D(texObj, x, y);



```
__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?

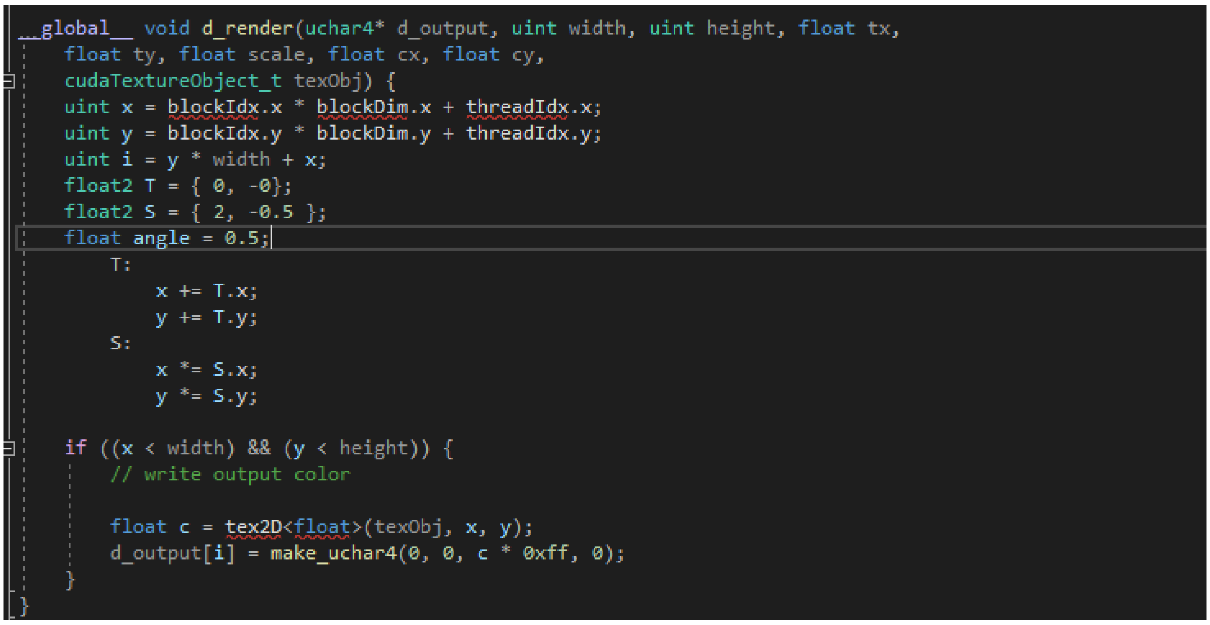
Question:

Demonstrate Image Rotation

Solution:

Rotate the image

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



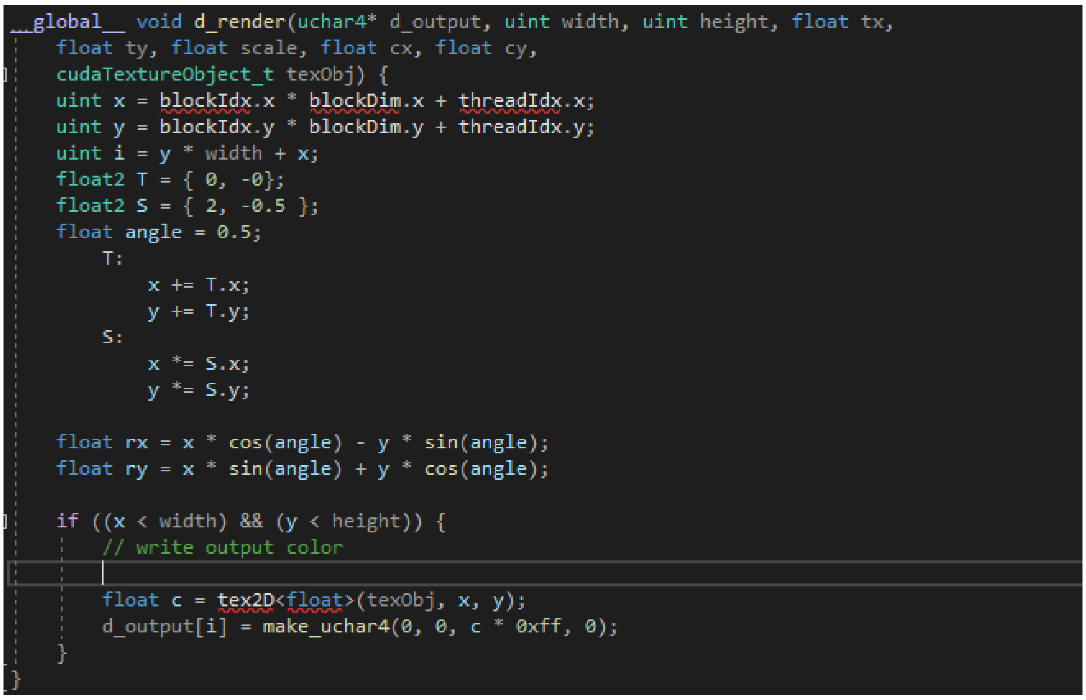
```
__global__ void d_render(uchar4* d_output, uint width, uint height, float tx,
    float ty, float scale, float cx, float cy,
    cudaTextureObject_t texObj) {
    uint x = blockIdx.x * blockDim.x + threadIdx.x;
    uint y = blockIdx.y * blockDim.y + threadIdx.y;
    uint i = y * width + x;
    float2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

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

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

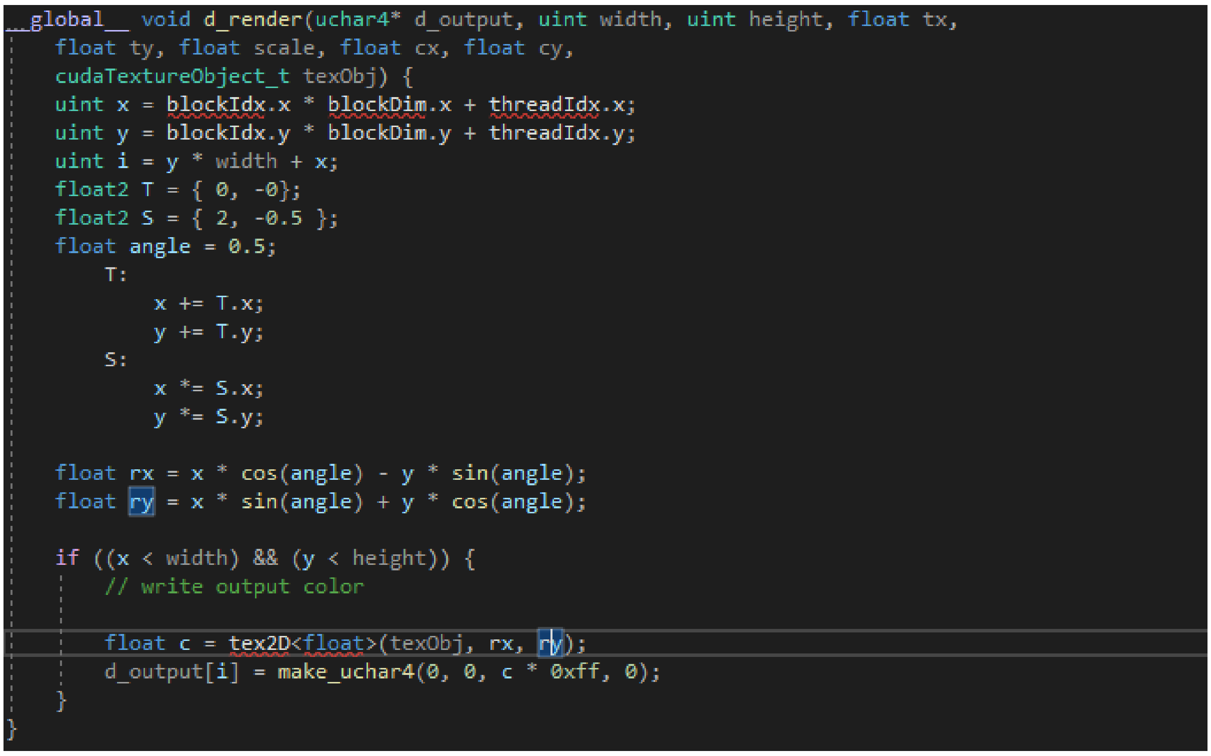
$$float\ ry = x * sin(angle) + y * cos(angle);$$



```
__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

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

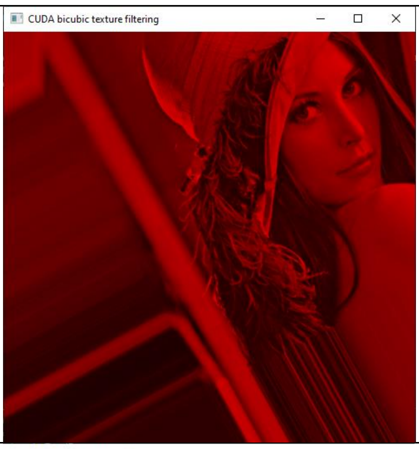
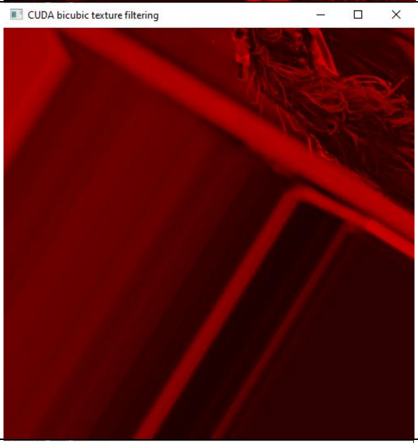
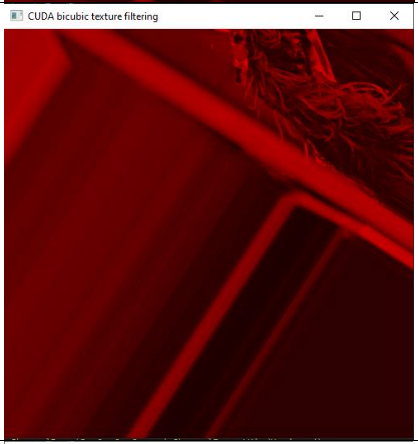
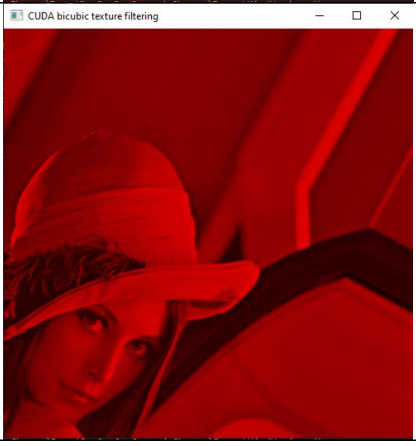


```
__global__ void d_render(uchar4* d_output, uint width, uint height, float tx,
    float ty, float scale, float cx, float cy,
    cudaTextureObject_t texObj) {
    uint x = blockIdx.x * blockDim.x + threadIdx.x;
    uint y = blockIdx.y * blockDim.y + threadIdx.y;
    uint i = y * width + x;
    float2 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;`	 CUDA bicubic texture filtering
`float angle = 45;`	 CUDA bicubic texture filtering
`float angle = 1;`	 CUDA bicubic texture filtering
`float angle = -0.5;`	 CUDA bicubic texture filtering

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:

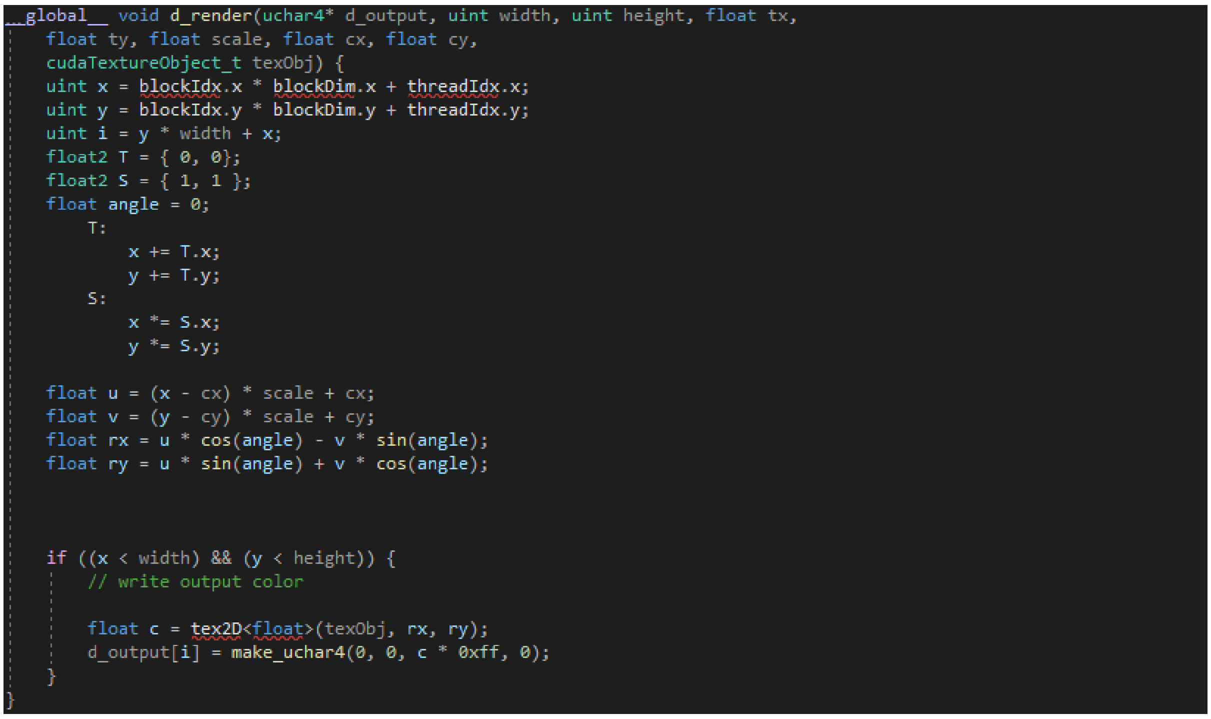
Demonstrate scaling by position

Solution:

Implement the following struct:

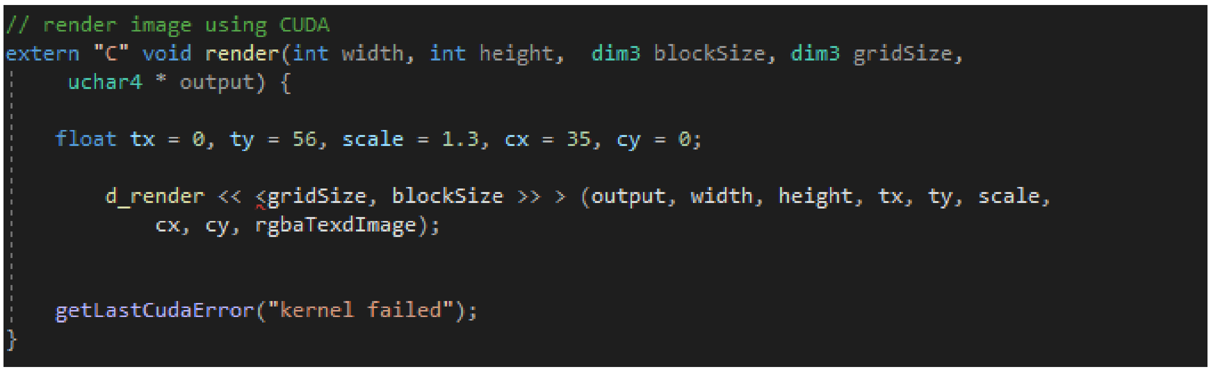
$$float\ u = (x - cx) * scale + cx;$$

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



```
__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);
    }
}
```

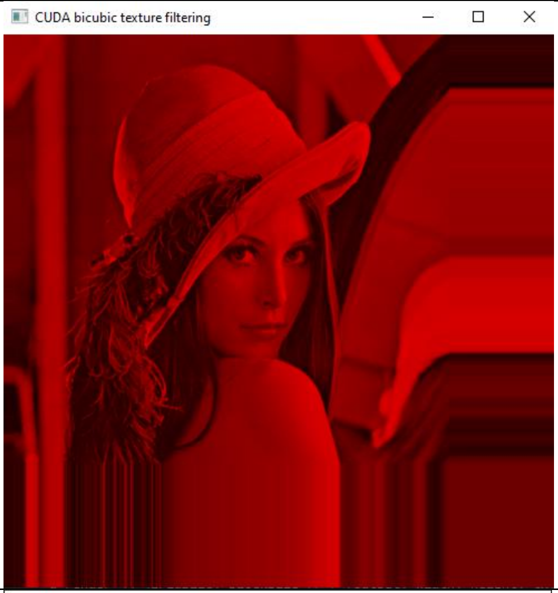
Now the image can be edited by adjusting the values passed into the kernel as shown below:

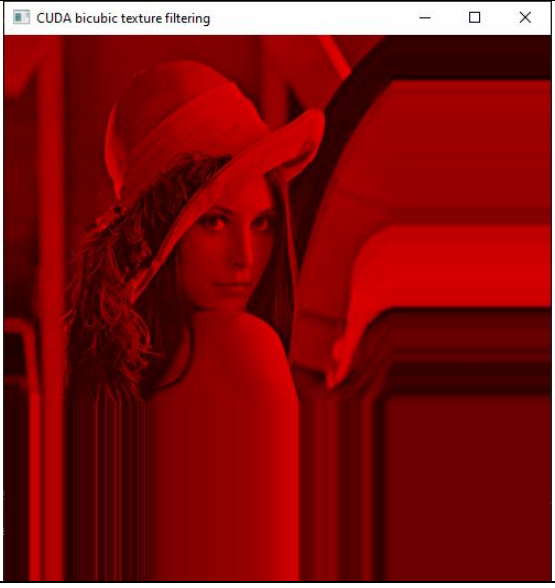


```
// render image using CUDA
extern "C" void render(int width, int height,  dim3 blockSize, dim3 gridSize,
     uchar4 * output) {

    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:

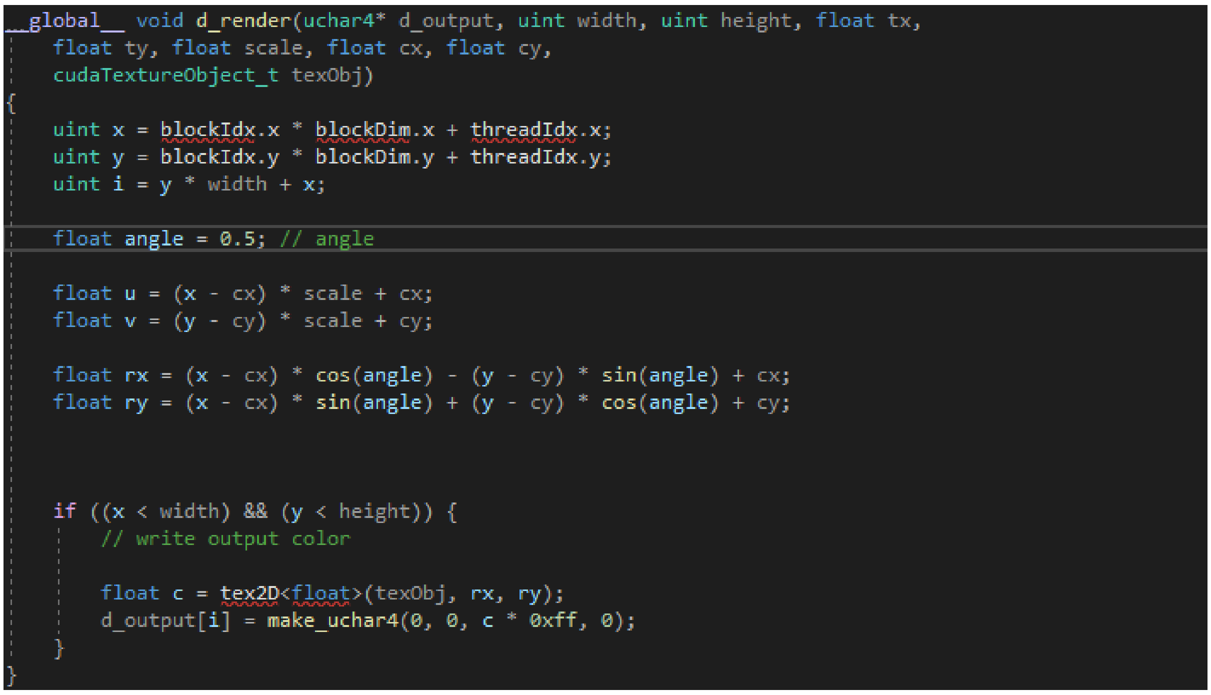
None.

Question:

Demonstrate rotation about a centre point

Solution:

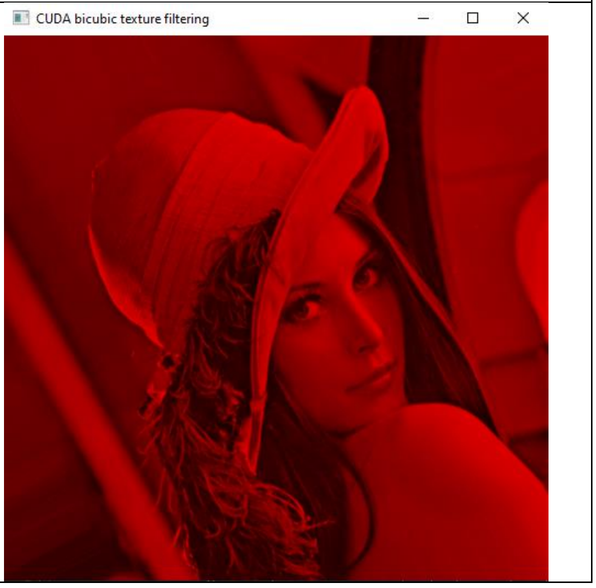
Modify the rotation code to incorporate the passed in centre point values.



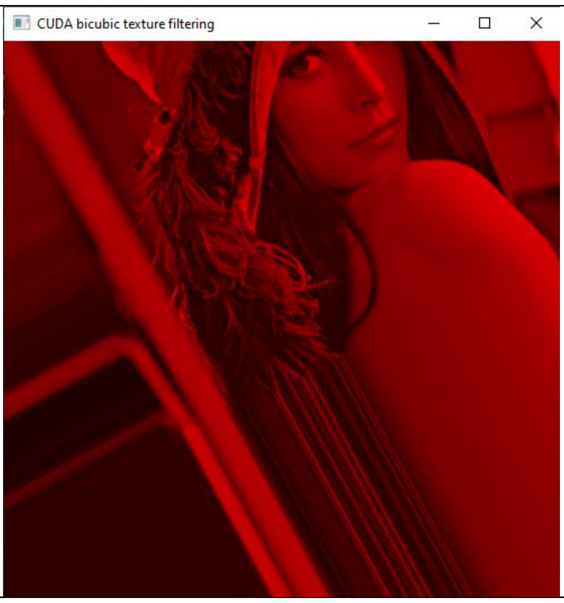
```
__global__ void d_render(uchar4* d_output, uint width, uint height, float tx,
    float ty, float scale, float cx, float cy,
    cudaTextureObject_t texObj)
{
    uint x = blockIdx.x * blockDim.x + threadIdx.x;
    uint y = blockIdx.y * blockDim.y + threadIdx.y;
    uint i = y * width + x;

    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:

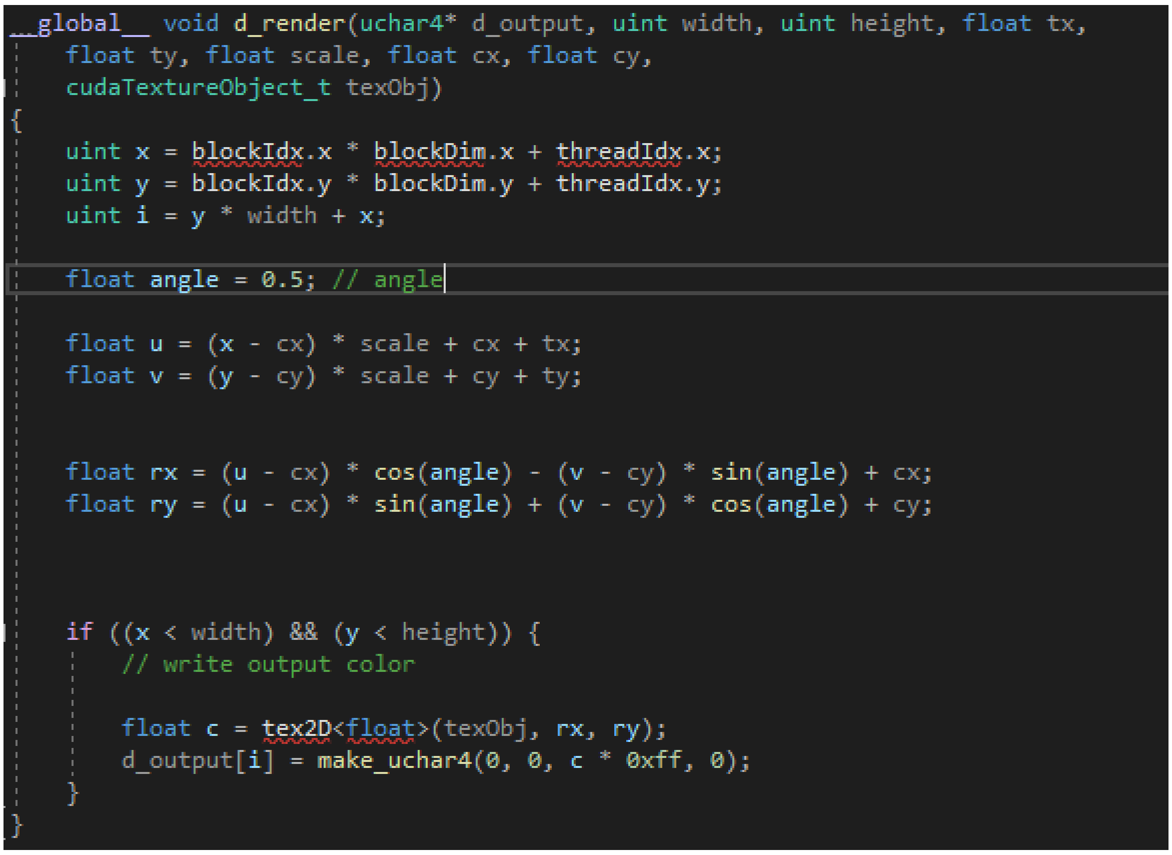
None.

Question:

Demonstrate simplified translation implementation.

Solution:

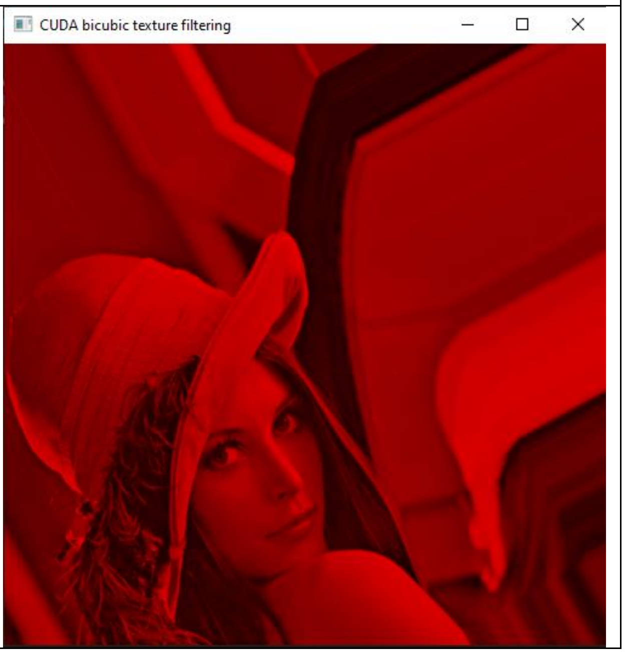
The below code implementation will translate the image based on the passed in bvariable to the kernel



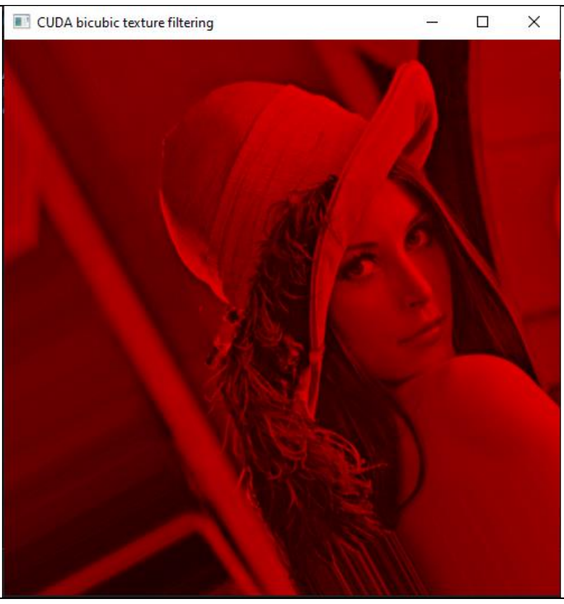
```
__global__ void d_render(uchar4* d_output, uint width, uint height, float tx,
    float ty, float scale, float cx, float cy,
    cudaTextureObject_t texObj)
{
    uint x = blockIdx.x * blockDim.x + threadIdx.x;
    uint y = blockIdx.y * blockDim.y + threadIdx.y;
    uint i = y * width + x;

    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.

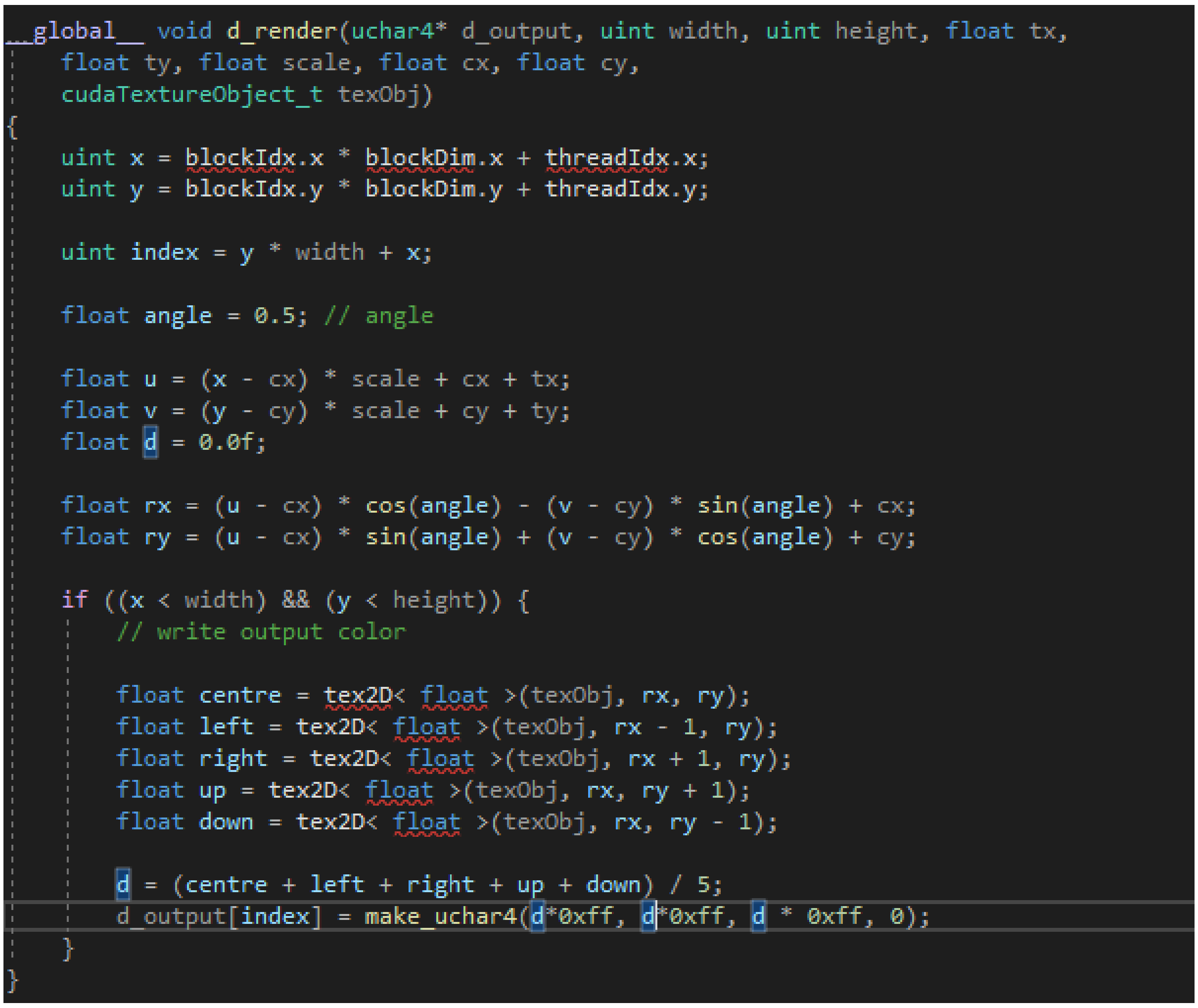
Exercise 4. Image smoothing

Question:

Demonstrate Image smoothing.

Solution:

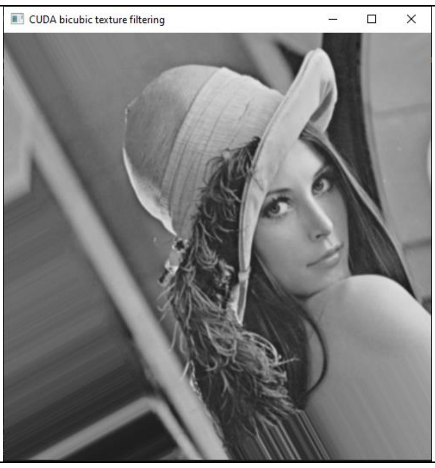
The below code implements image smoothing using an order 1 square neighbour for reference



```
__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.

600086 Lab Book

Week 5 – Lab B

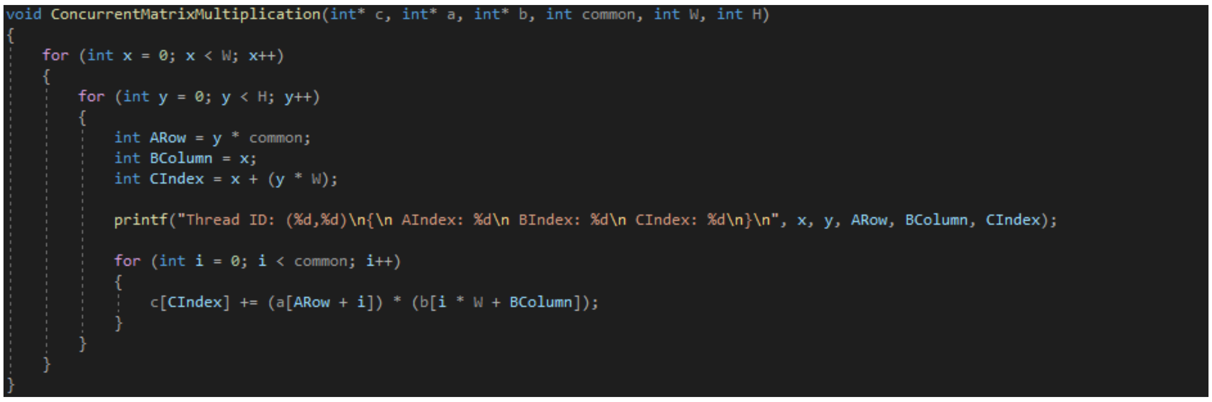
Date: 3rd March 2022

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

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

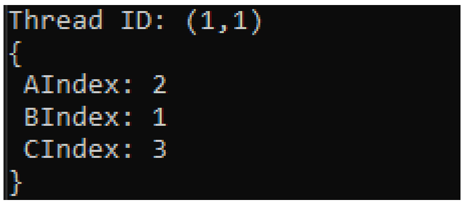
Solution:

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{\n AIndex: %d\n BIndex: %d\n CIndex: %d\n}\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



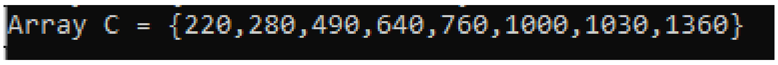
```
Thread ID: (1,1)
{
 AIndex: 2
 BIndex: 1
 CIndex: 3
}
```

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:

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

Sample output:

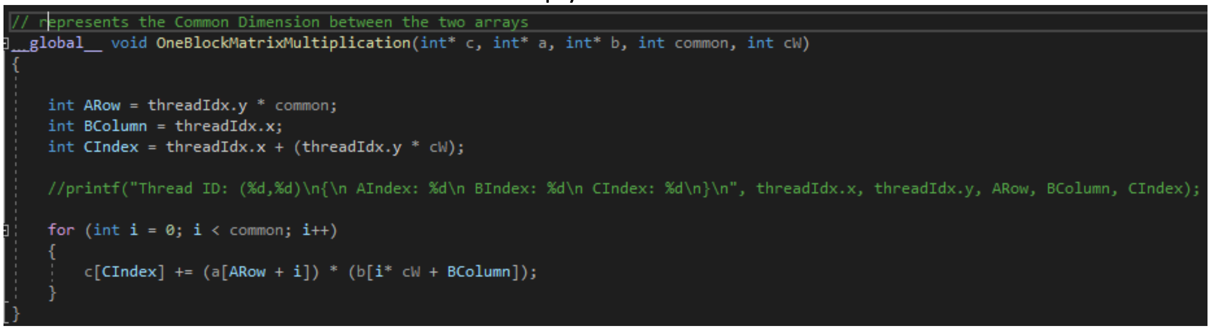


```
Array C = {220,280,490,640,760,1000,1030,1360}
```

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

Solution:

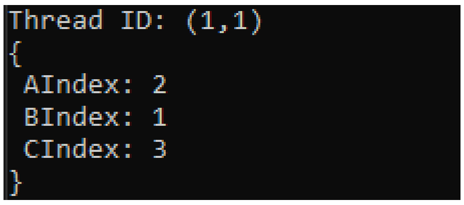
1. Modified the Add kernel Function to multiply the matrixes as follows



```
// 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



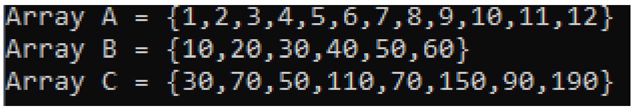
```
Thread ID: (1,1)
{
 AIndex: 2
 BIndex: 1
 CIndex: 3
}
```

   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:

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

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 threadId.x and threadId.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

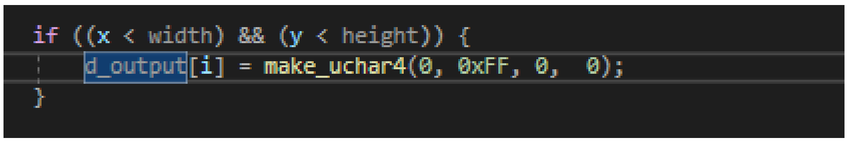
Date: 10th Mar 2022

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:

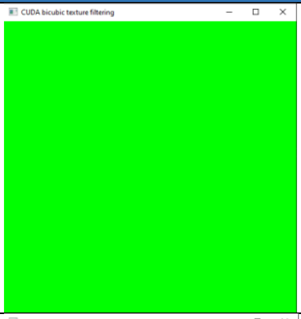
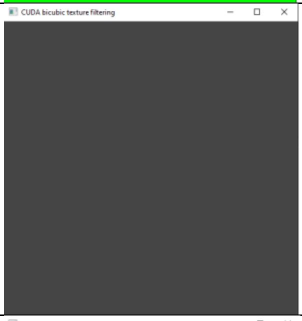
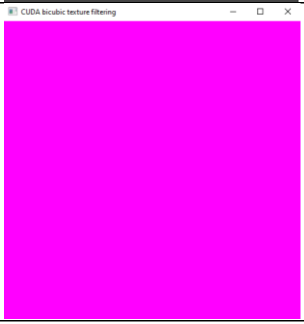


```
if ((x < width) && (y < height)) {
    d_output[i] = make_uchar4(0, 0xFF, 0,  0);
}
```

Test data:

N/A

Sample output:

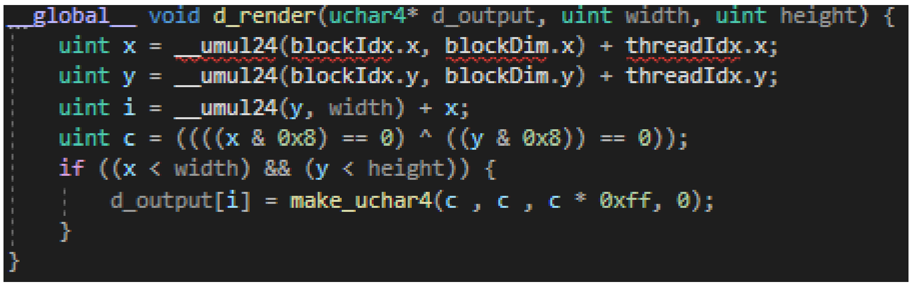
Input	expectation	Output
`(0, 0xFF, 0, 0)`	Green	
`(0x45, 0xF45, 0x45, 0)`	Grey	
`(0, 0xFF, 0, 0)`	Fuchsia	

Exercise 2. Drawing a checkboard in CUDA

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

Solution:

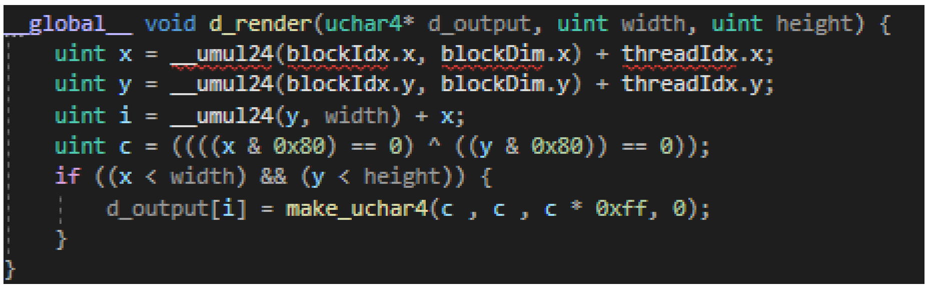
1. Edit the d_render( ) method to draw a checkboard



```
__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

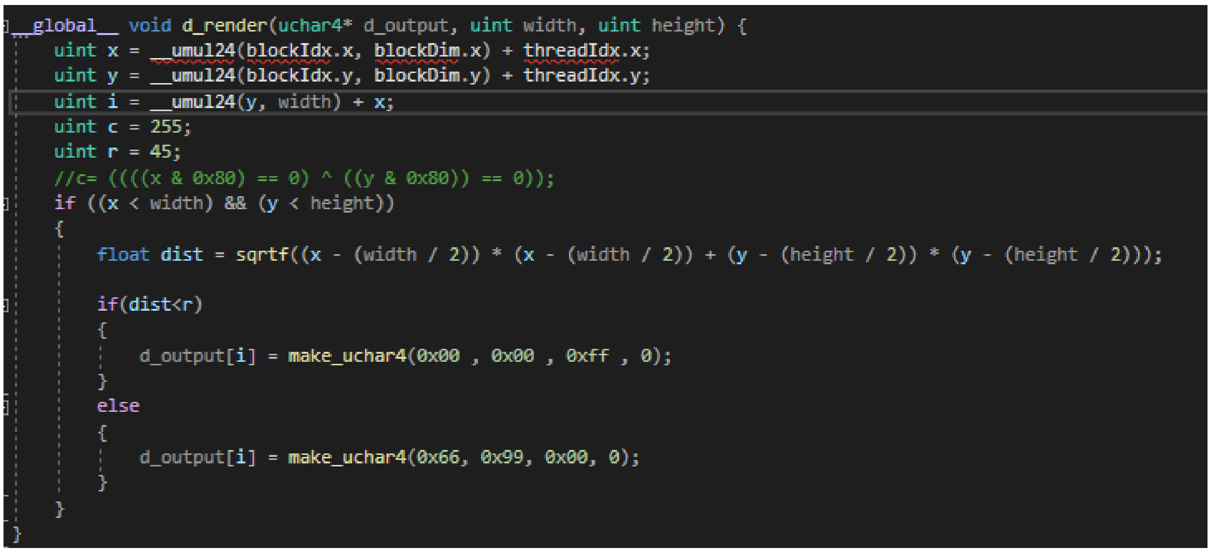
2. Modify the code to draw a checkboard with much larger red-blocks



```
__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.

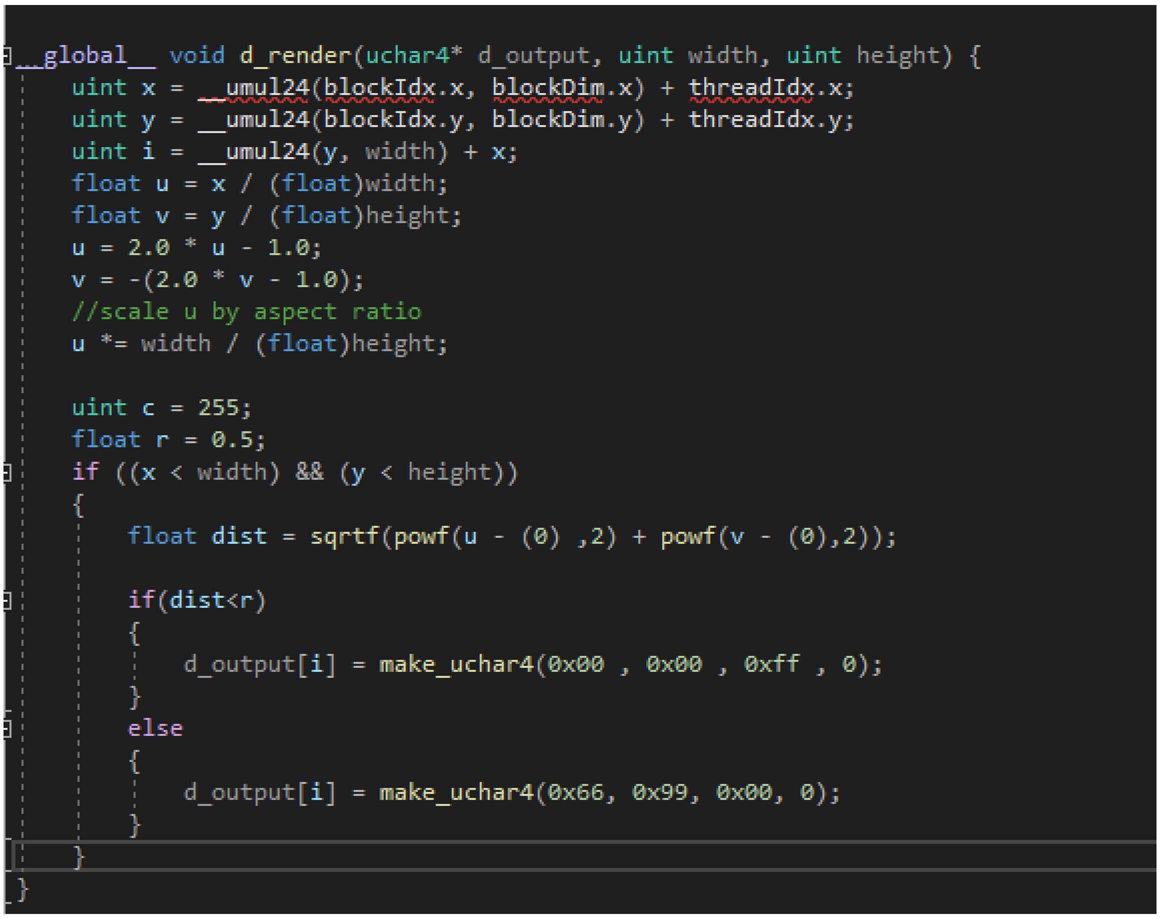
3. Further modify your code to draw a red disc in the middle of the image of a red disc:



```
__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]x[-1,1]



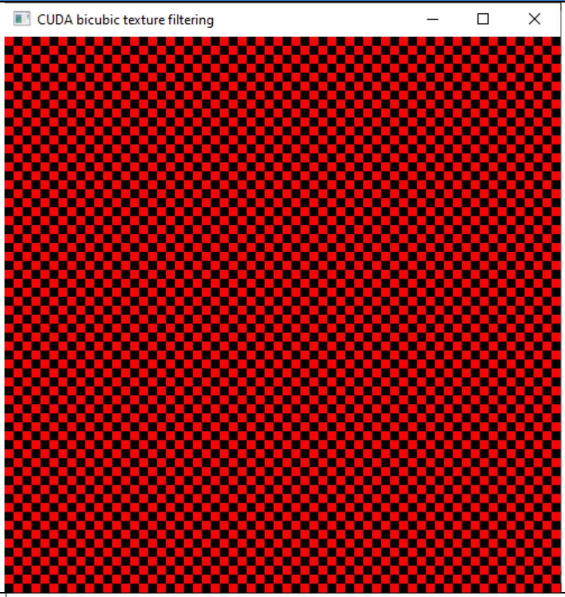
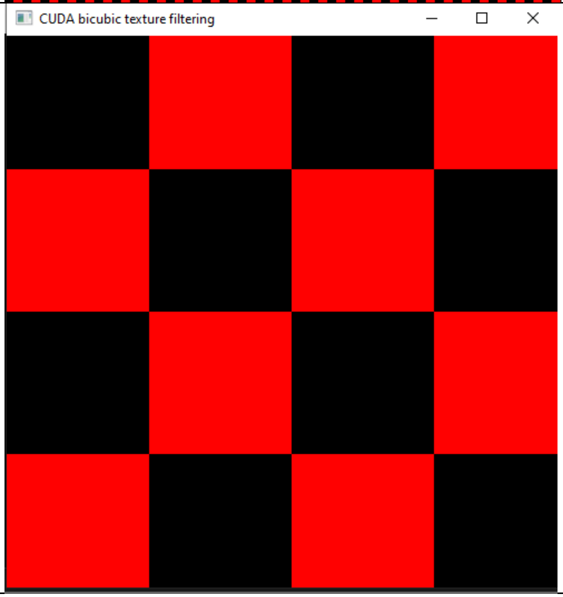
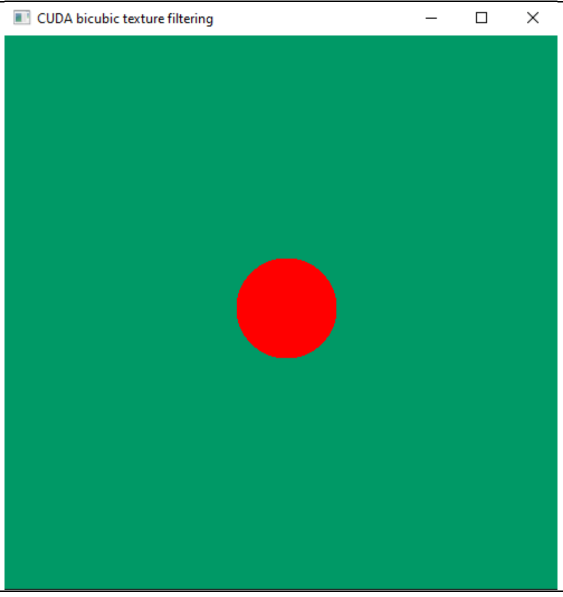
```
__global__ void d_render(uchar4* d_output, uint width, uint height) {
    uint x = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
    uint y = __umul24(blockIdx.y, blockDim.y) + threadIdx.y;
    uint i = __umul24(y, width) + x;
    float u = x / (float)width;
    float v = y / (float)height;
    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);
        }
    }
}
```

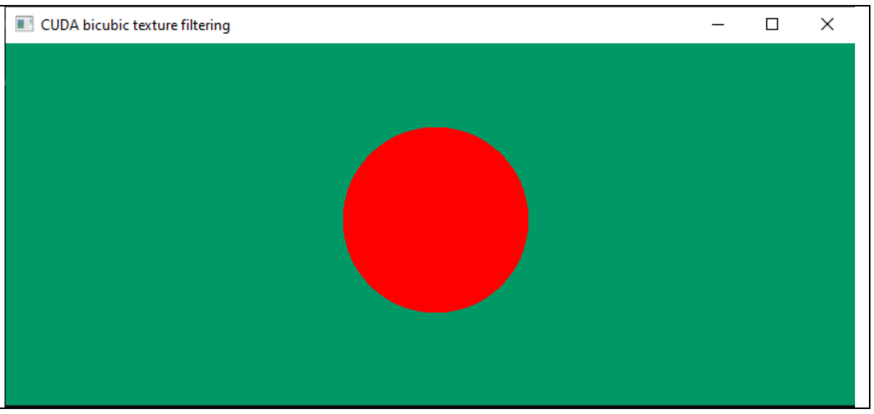
I Added in a translation for the pixel location represented by u and v and then added a acle translation to ensure the resultant image matched the aspect ratio of the window to prevent distortion. See Sample Output Ref 4 for the result.

Test data:

N/A

Sample output:

REF	Output
1	 CUDA bicubic texture filtering
2	 CUDA bicubic texture filtering
3	 CUDA bicubic texture filtering

4	

Reflection:

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

Metadata:

Further information:

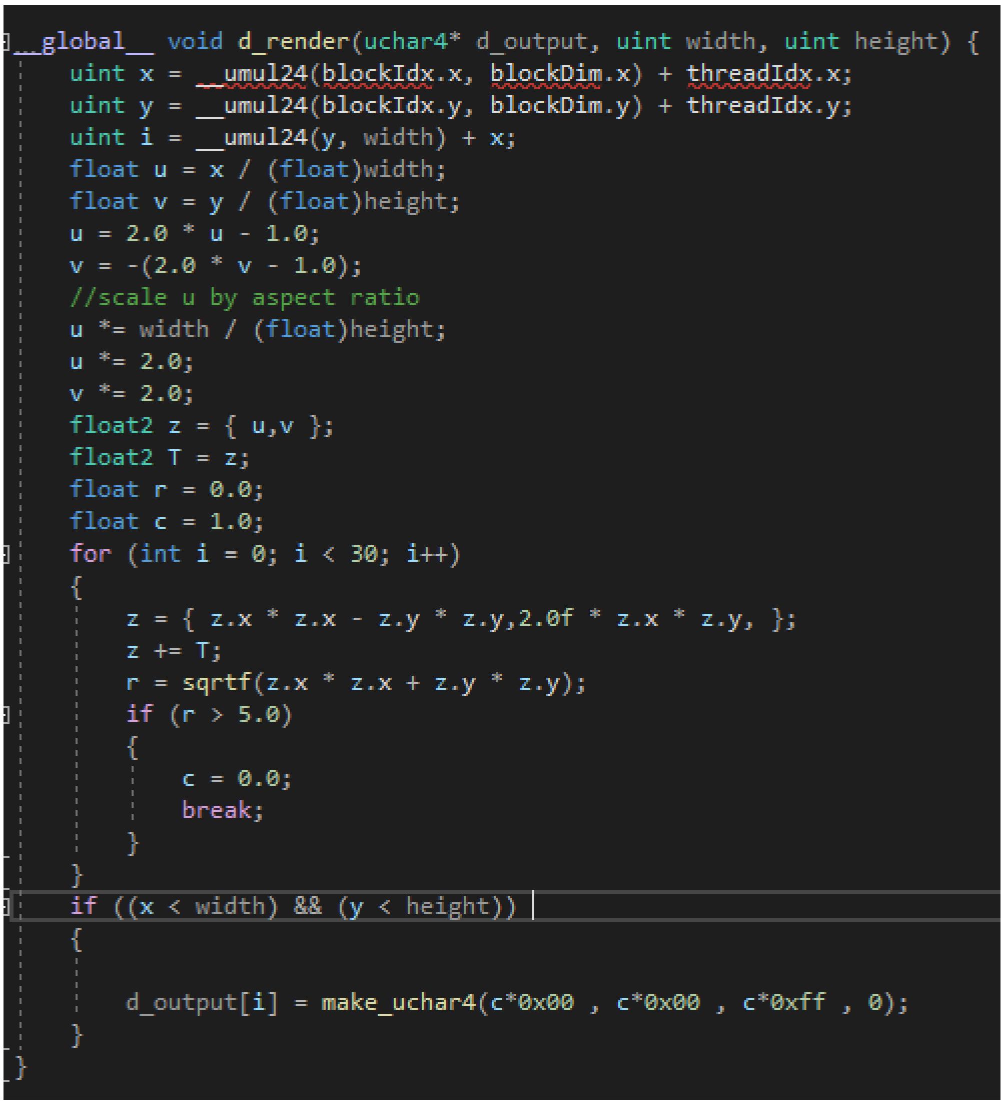
is this similar to how vector graphics are created?

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

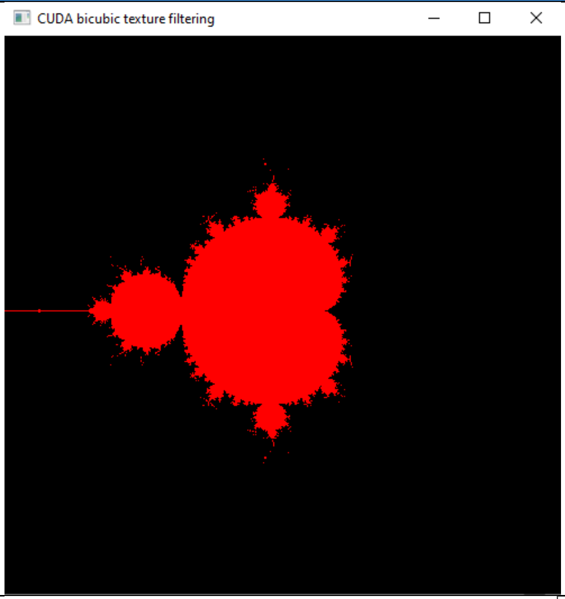
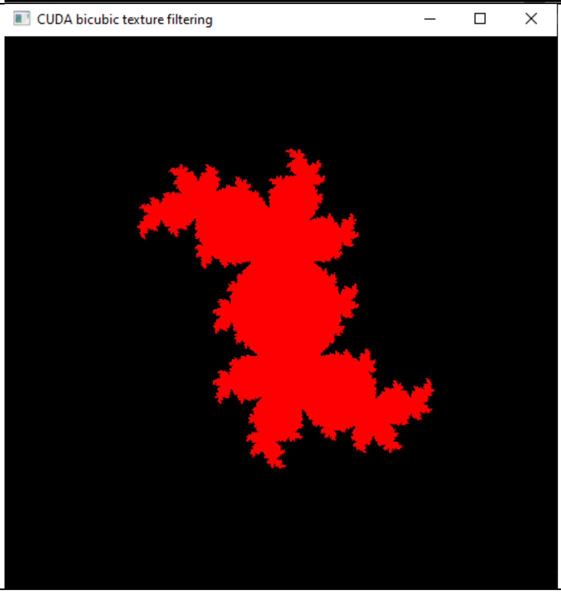
Test data:

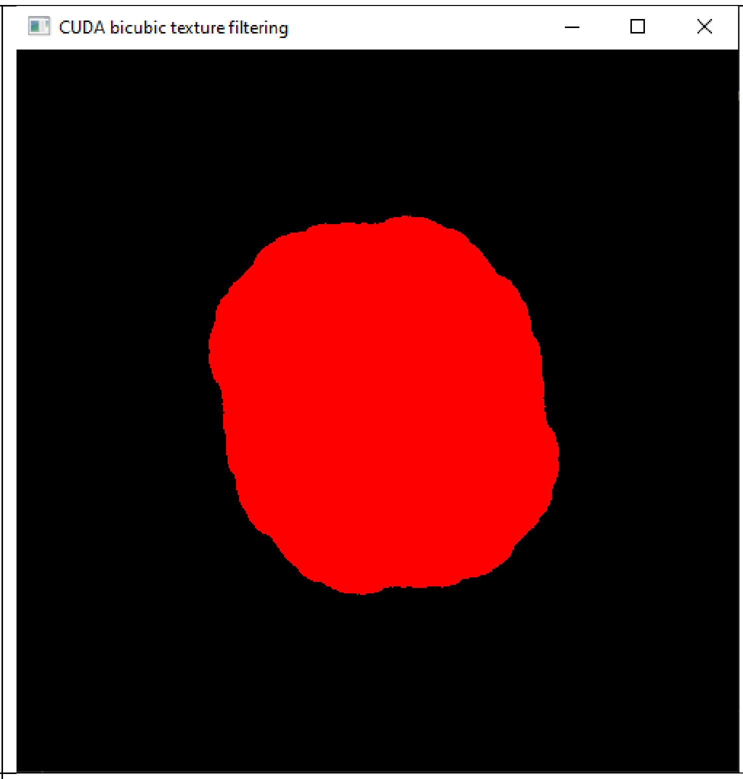
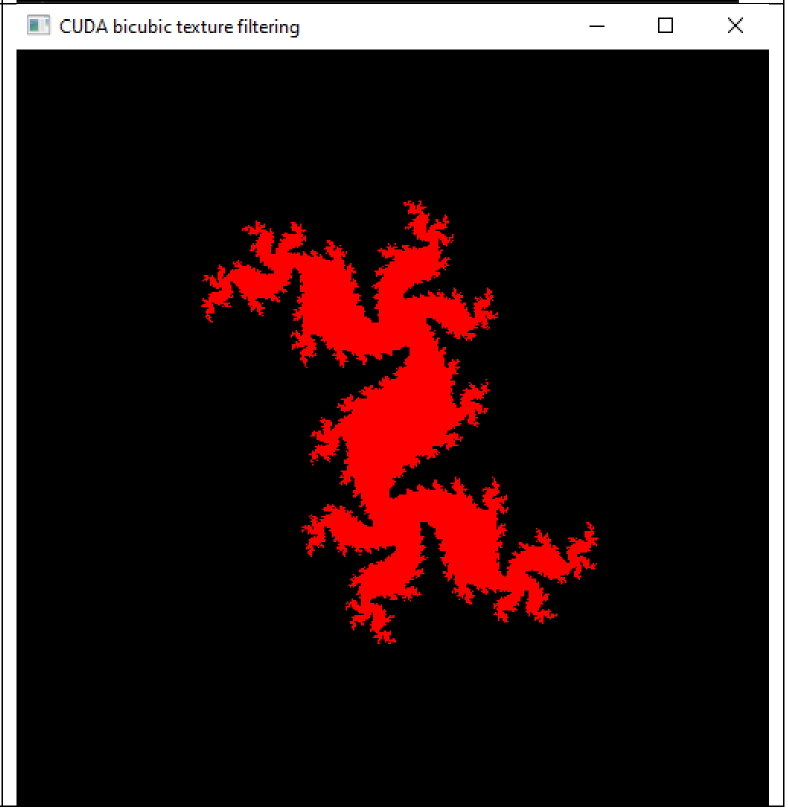
T = {0.25, 0.5}

T = {0.1, 0.1}

T = {0.3, 0.5}

Sample output:

ref	Output
1	 CUDA bicubic texture filtering
2	 CUDA bicubic texture filtering

3	
4	

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

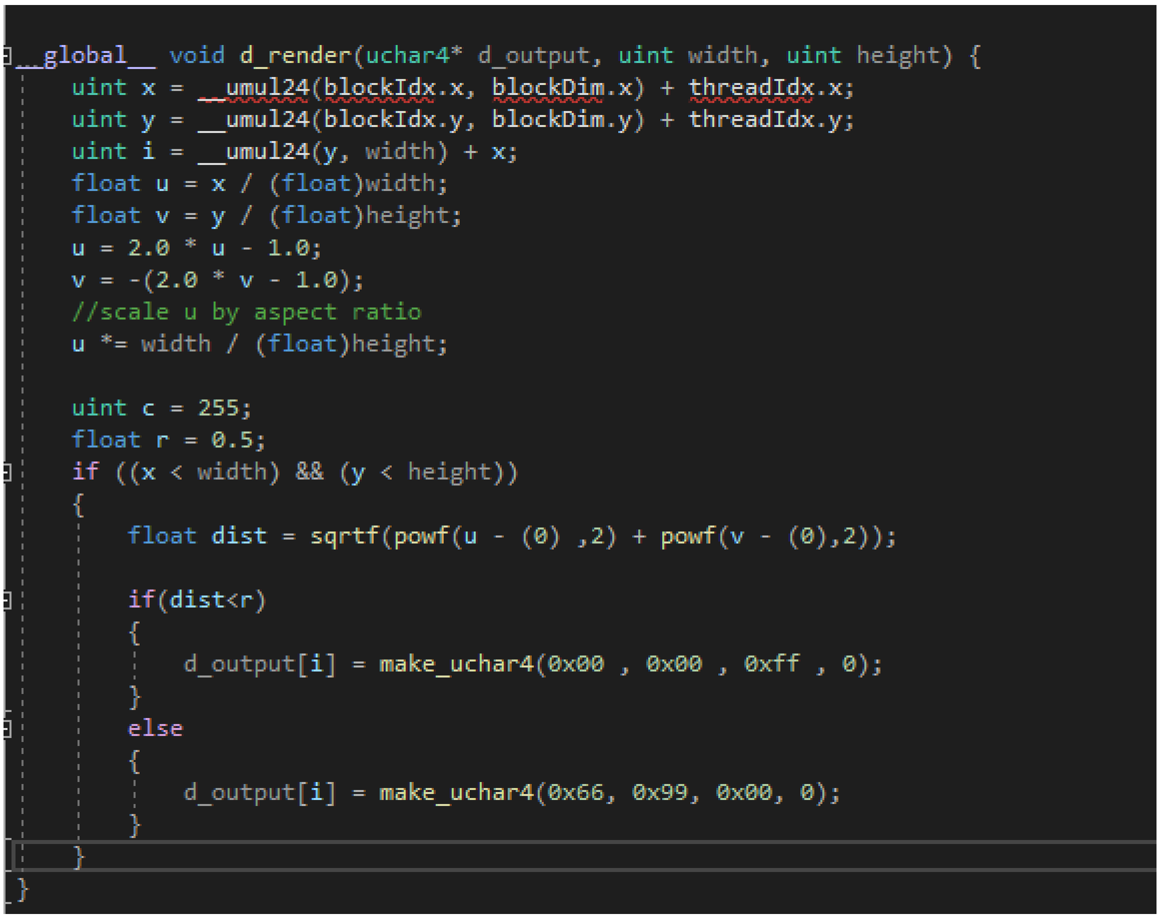
Date: 24th Mar 2022

Exercise 1. Drawing based on a canvas of size [-1, 1]x[-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]x[-1,1]



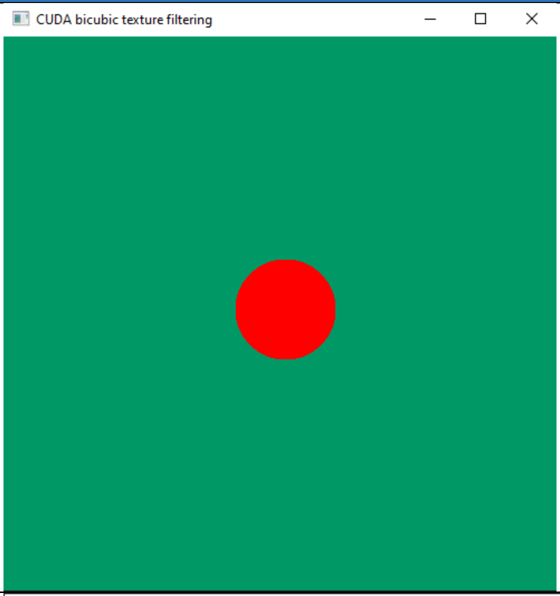
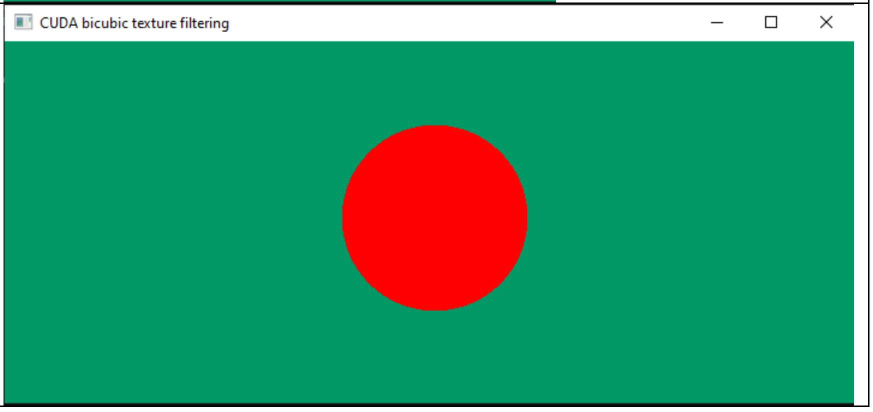
```
__global__ void d_render(uchar4* d_output, uint width, uint height) {
    uint x = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
    uint y = __umul24(blockIdx.y, blockDim.y) + threadIdx.y;
    uint i = __umul24(y, width) + x;
    float u = x / (float)width;
    float v = y / (float)height;
    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	 CUDA bicubic texture filtering
2	 CUDA bicubic texture filtering

Reflection:

none

Metadata:

Further information:

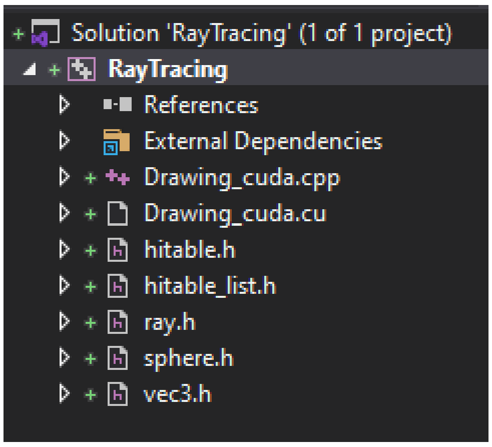
none

Exercise 2. Write a simple ray caster

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

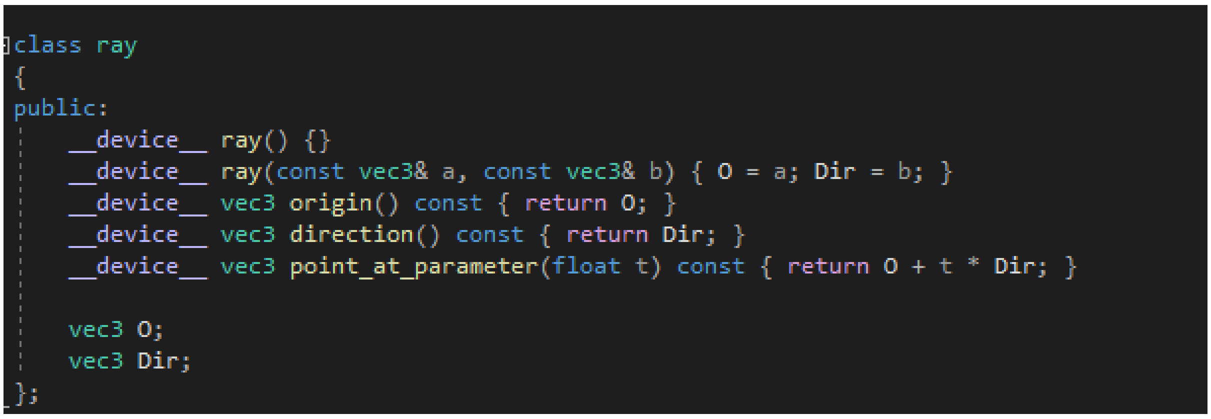
Solution:

1. Add the necessary header files to the project



```
Solution 'RayTracing' (1 of 1 project)
  RayTracing
    References
    External Dependencies
    Drawing_cuda.cpp
    Drawing_cuda.cu
    hitable.h
    hitable_list.h
    ray.h
    sphere.h
    vec3.h
```

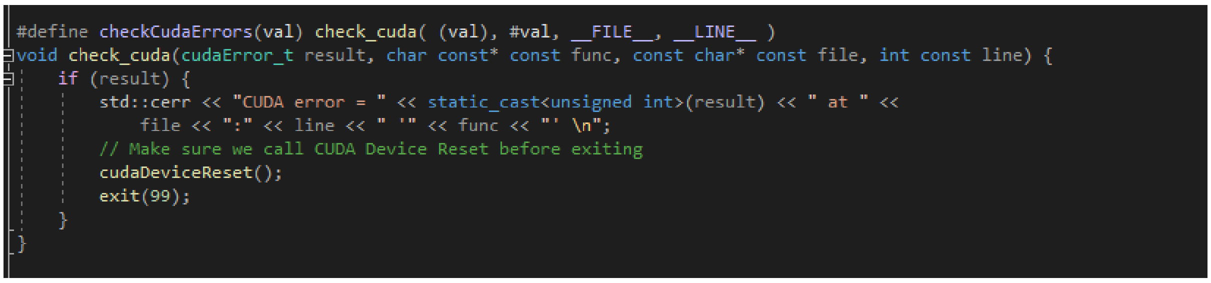
2. Change the variable names in the ray class to make them more meaningful for the current implementation



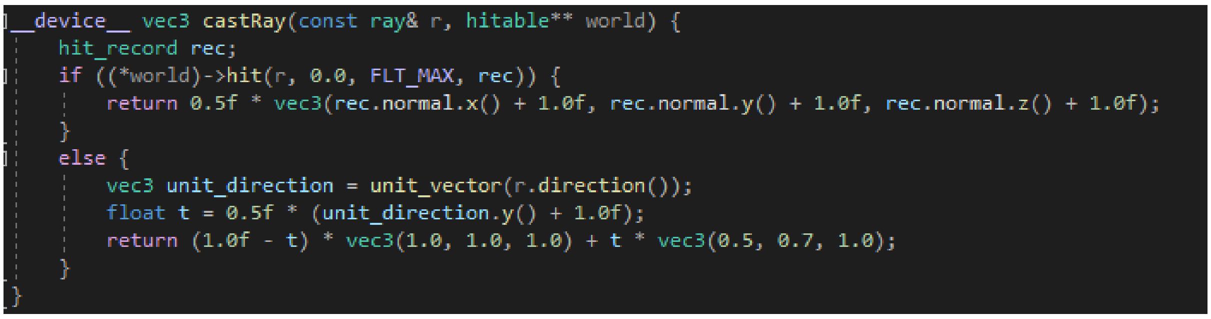
```
class ray
{
public:
    __device__ ray() {}
    __device__ ray(const vec3& a, const vec3& b) { O = a; Dir = b; }
    __device__ vec3 origin() const { return O; }
    __device__ vec3 direction() const { return Dir; }
    __device__ vec3 point_at_parameter(float t) const { return O + t * Dir; }

    vec3 O;
    vec3 Dir;
};
```

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

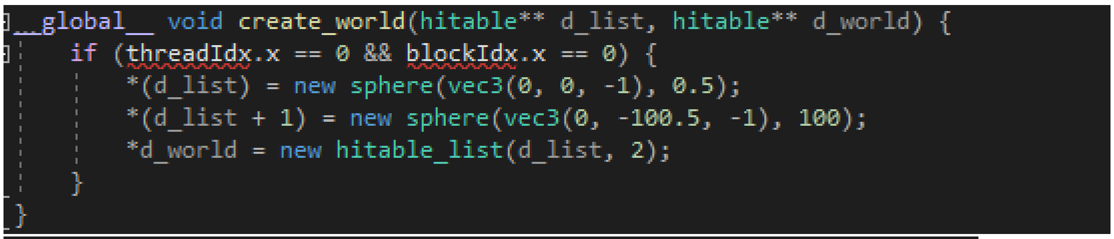
3. Implement the following functions cuda_check_error, castRay, create_world and free_world



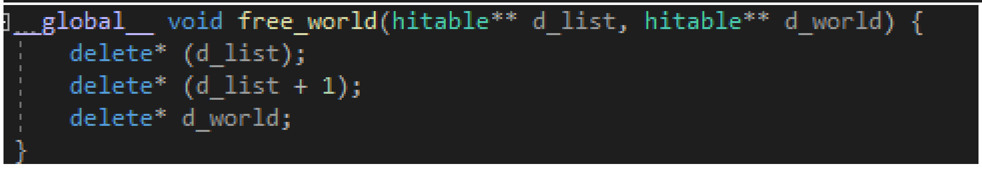
```
#define checkCudaErrors(val) check_cuda( (val), #val, __FILE__, __LINE__ )
void check_cuda(cudaError_t result, char const* const func, const char* const file, int const line) {
    if (result) {
        std::cerr << "CUDA error = " << static_cast<unsigned int>(result) << " at " <<
            file << ":" << 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);
    }
}
```

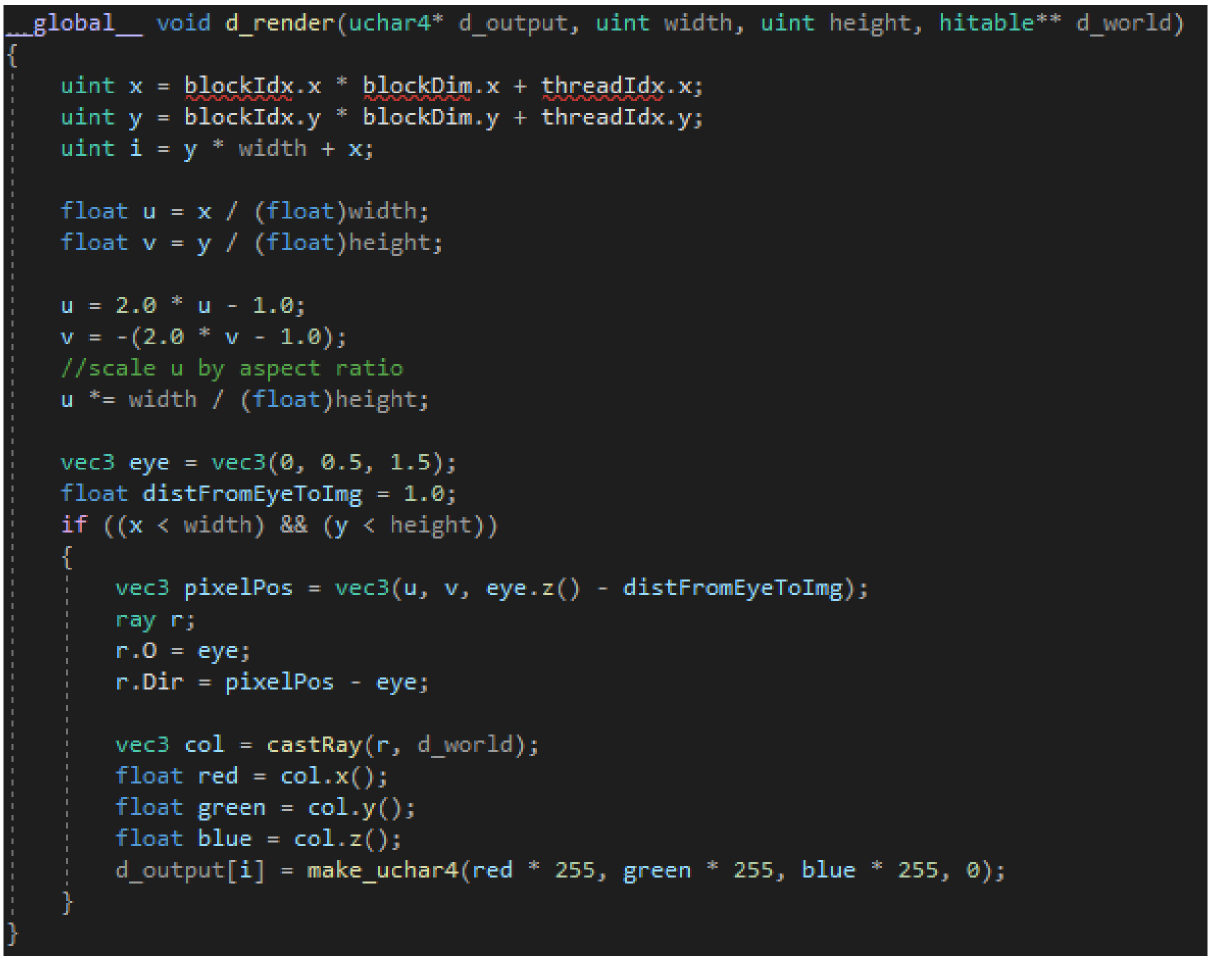


```
__global__ void create_world(hitable** d_list, hitable** d_world) {
    if (threadIdx.x == 0 && blockIdx.x == 0) {
        *(d_list) = new sphere(vec3(0, 0, -1), 0.5);
        *(d_list + 1) = new sphere(vec3(0, -100.5, -1), 100);
        *d_world = new hitable_list(d_list, 2);
    }
}
```



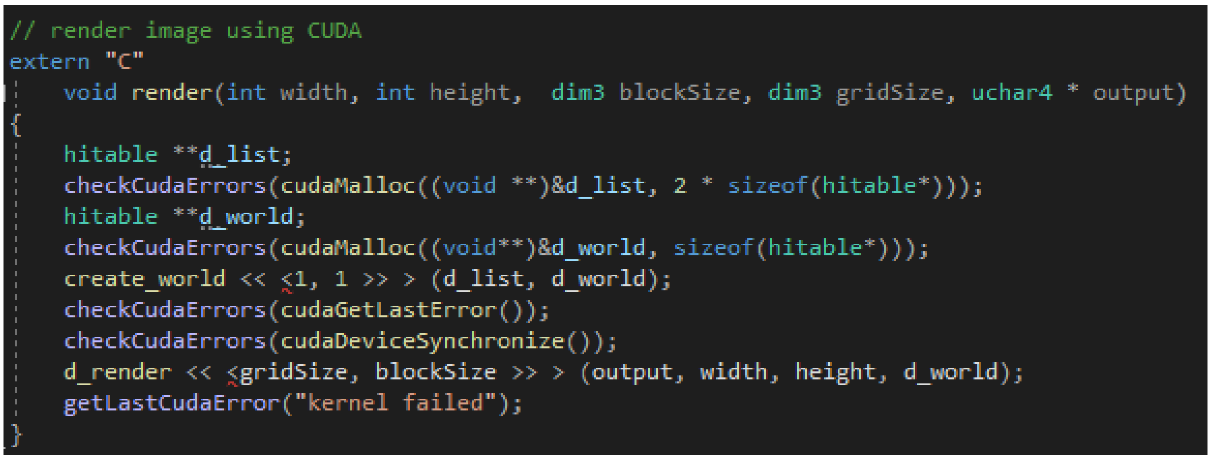
```
__global__ void free_world(hitable** d_list, hitable** d_world) {
    delete* (d_list);
    delete* (d_list + 1);
    delete* d_world;
}
```

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



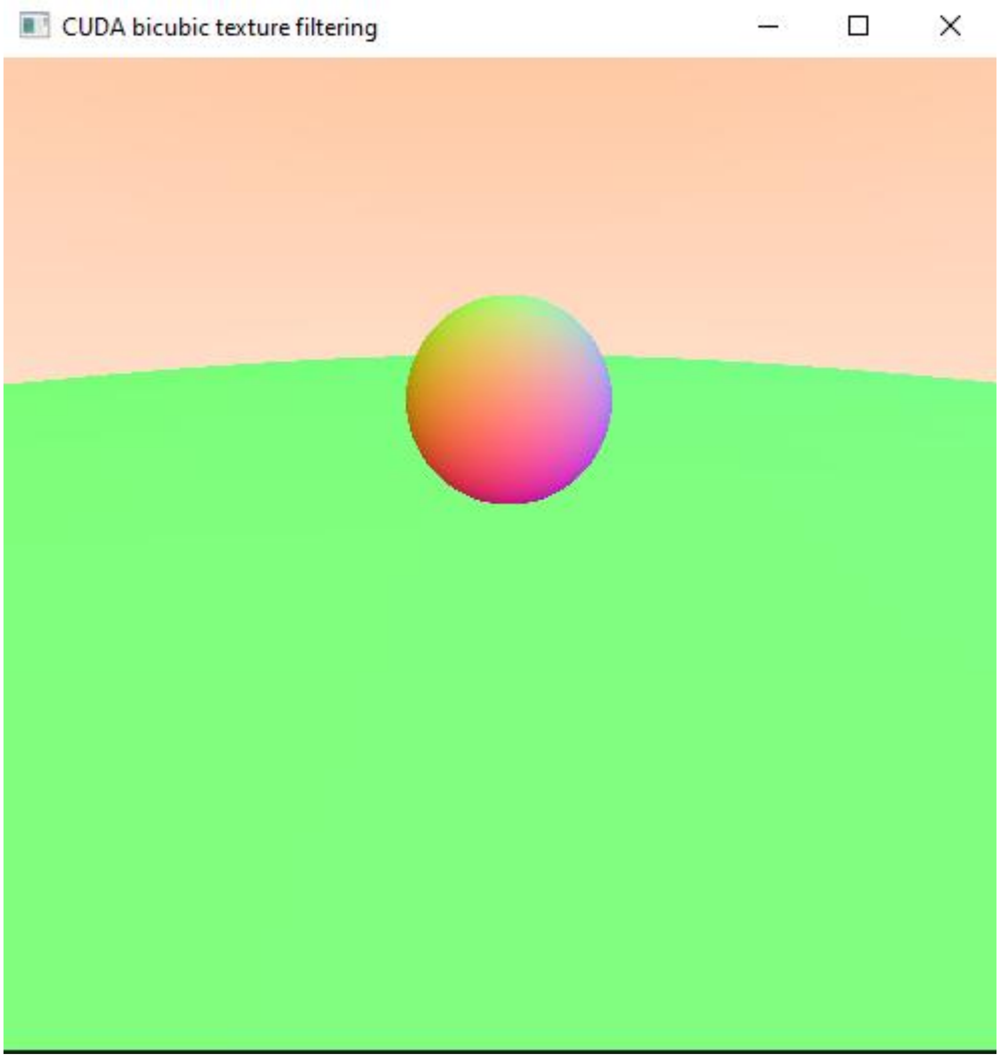
```
// render image using CUDA
extern "C"
    void render(int width, int height,  dim3 blockSize, dim3 gridSize, uchar4 * output)
{
    hitable **d_list;
    checkCudaErrors(cudaMalloc((void **)&d_list, 2 * sizeof(hitable*)));
    hitable **d_world;
    checkCudaErrors(cudaMalloc((void**)&d_world, sizeof(hitable*)));
    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:

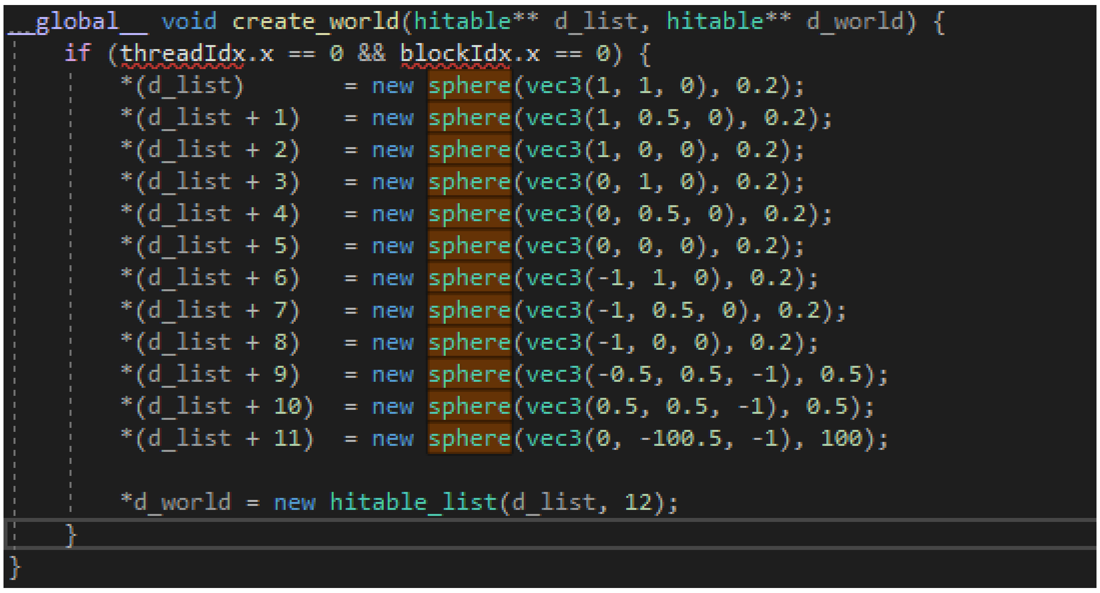
The task above was fairly easy to follow unsure why my red and blue values seem to have switched making my sky tinged red rather than the blue in the example.

Metadata:

Exercise 3. Adding multiple spheres to the ray caster

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

Solution:



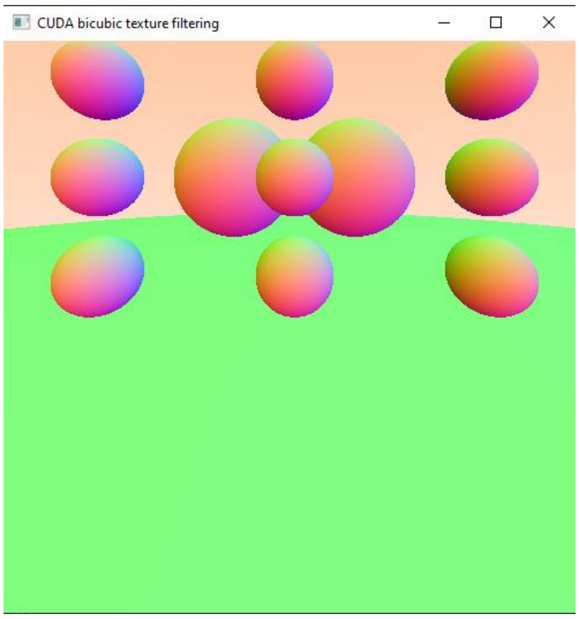
```
__global__ void create_world(hitable** d_list, hitable** d_world) {
    if (threadIdx.x == 0 && blockIdx.x == 0) {
        *(d_list)       = new sphere(vec3(1, 1, 0), 0.2);
        *(d_list + 1)   = new sphere(vec3(1, 0.5, 0), 0.2);
        *(d_list + 2)   = new sphere(vec3(1, 0, 0), 0.2);
        *(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

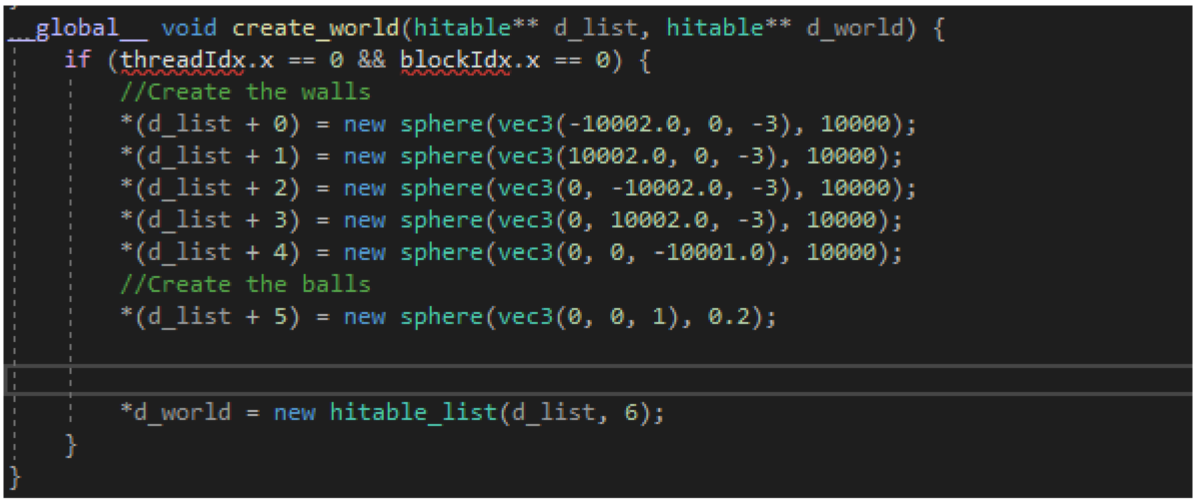
Date: 24th Mar 2022

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 sscreen

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.

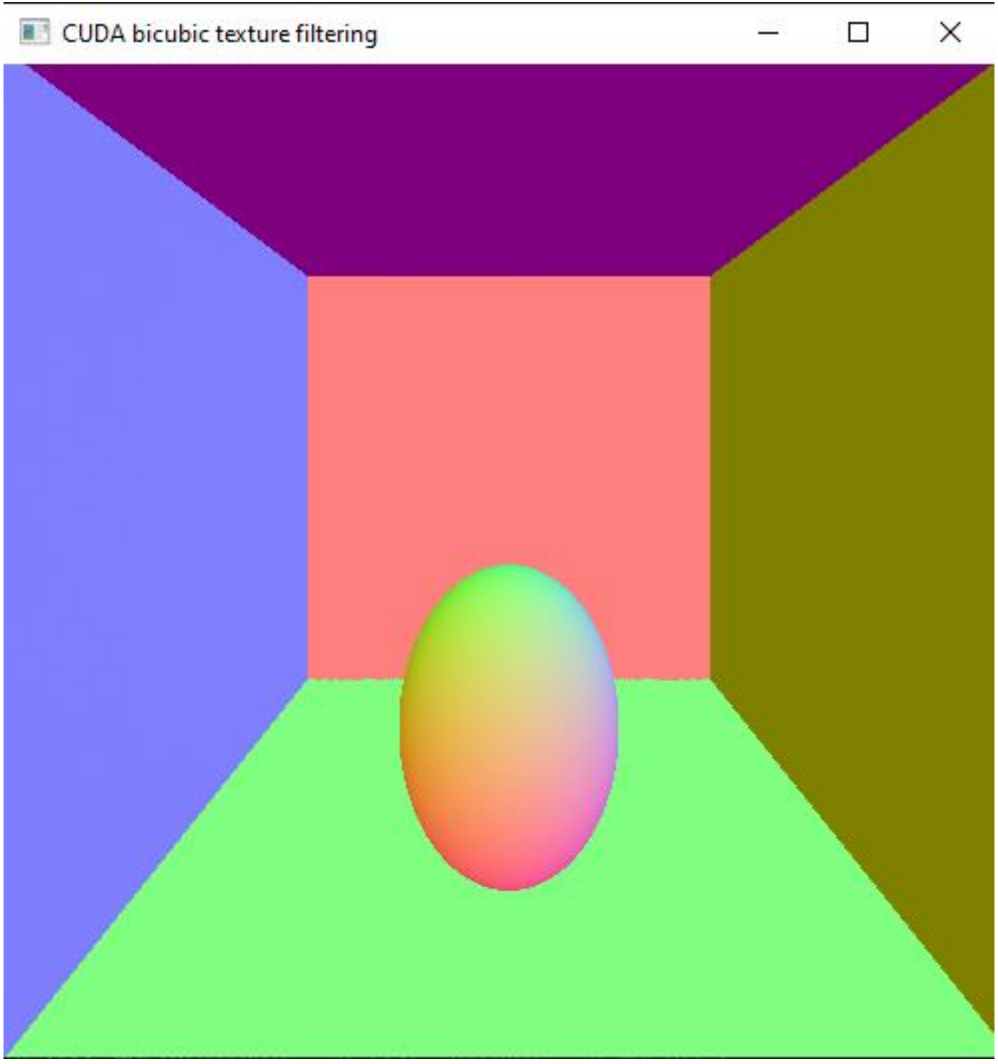


```
__global__ void create_world(hitable** d_list, hitable** d_world) {
    if (threadIdx.x == 0 && blockIdx.x == 0) {
        //Create the walls
        *(d_list + 0) = new sphere(vec3(-10002.0, 0, -3), 10000);
        *(d_list + 1) = new sphere(vec3(10002.0, 0, -3), 10000);
        *(d_list + 2) =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

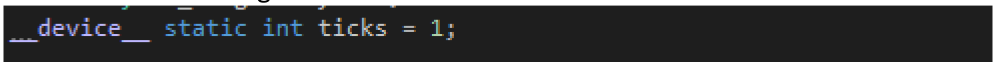
Further information:

Exercise 2. Free motion animation

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

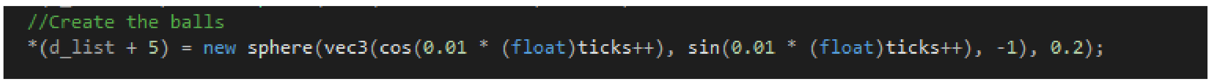
Solution:

Added a device wide global variable named tick



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

Changed the sphere to be drawn to the following



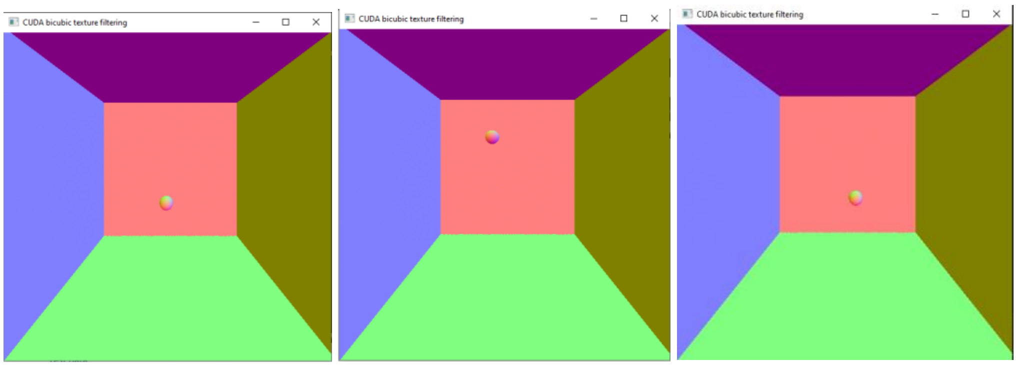
```
//Create the balls
*(d_list + 5) = new sphere(vec3(cos(0.01 * (float)ticks++), sin(0.01 * (float)ticks++), -1), 0.2);
```

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

Test data:

n/a

Sample output:


CUDA bicubic texture filtering
CUDA bicubic texture filtering
CUDA bicubic texture filtering

Reflection:

This was fairly perfunctory and following the lab made sense

Metadata:

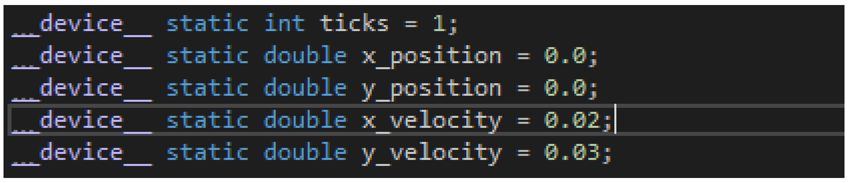
Exercise 3. Ball-box walls collision animation

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;

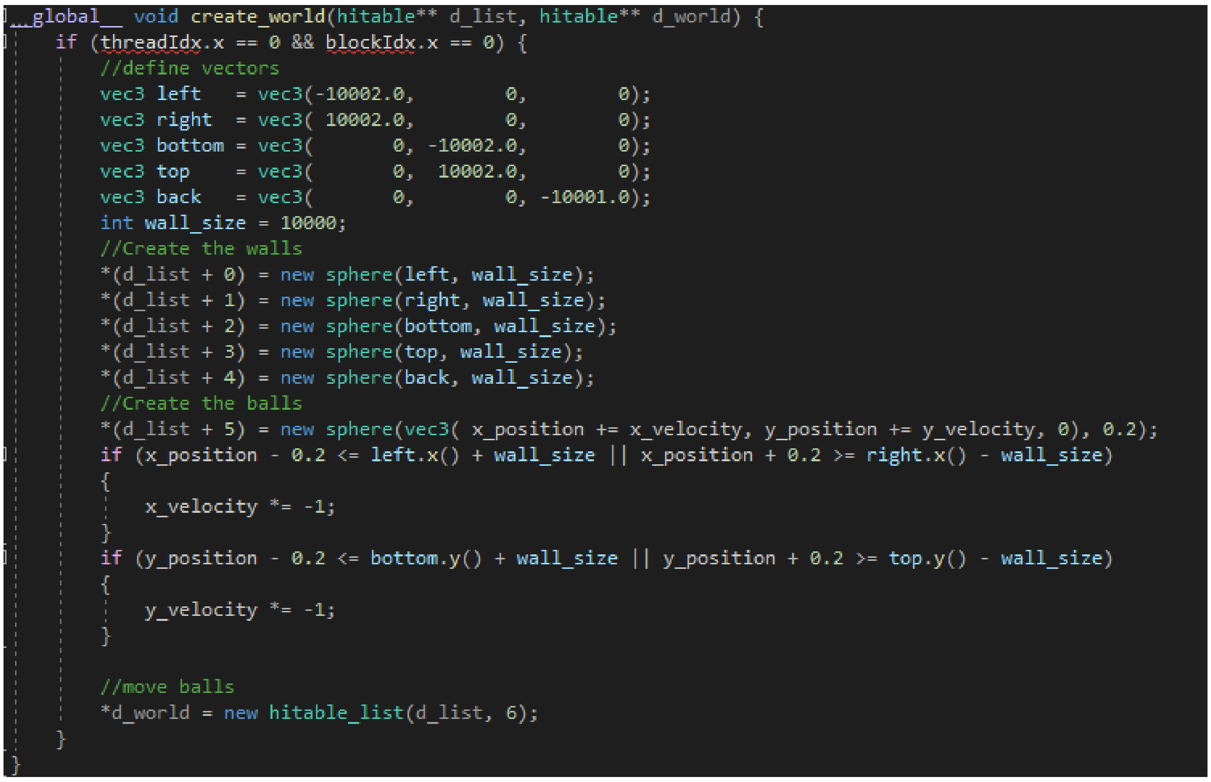
Solution:

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

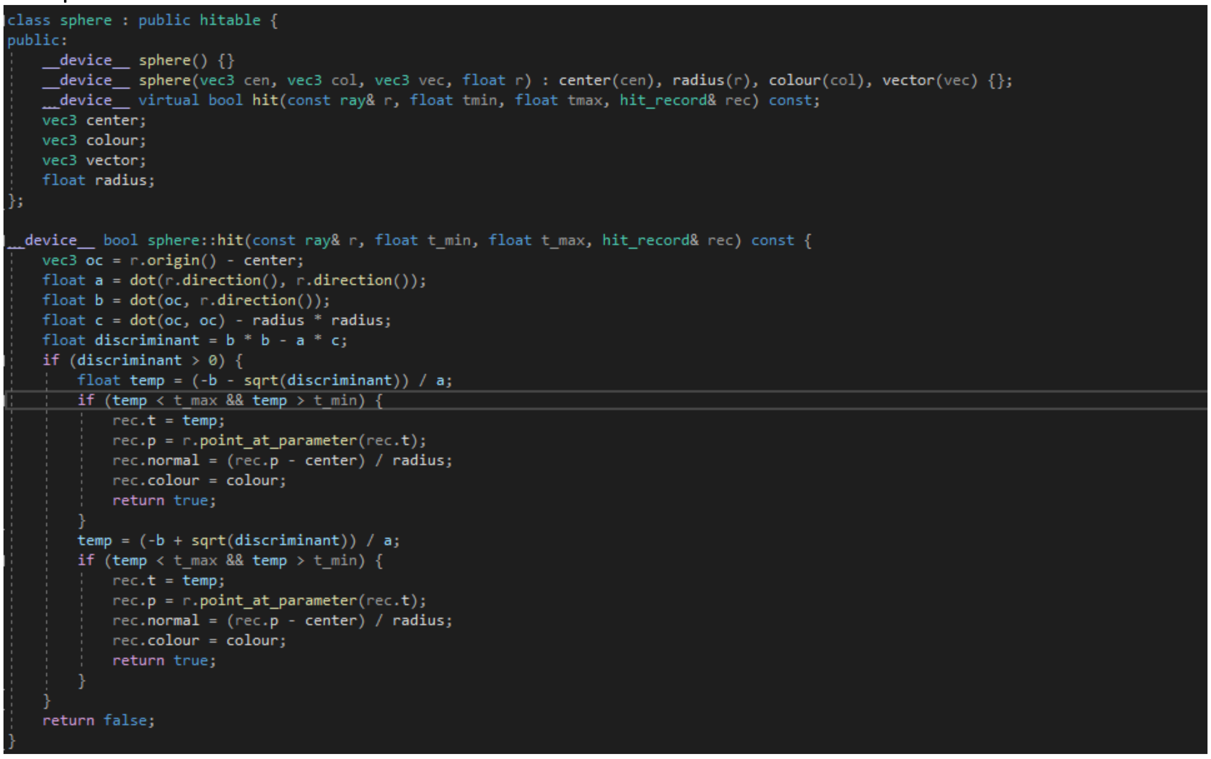


```
__global__ void create_world(hitable** d_list, hitable** d_world) {
    if (threadIdx.x == 0 && blockIdx.x == 0) {
        //define vectors
        vec3 left   = vec3(-10002.0,        0,        0);
        vec3 right  = vec3( 10002.0,        0,        0);
        vec3 bottom = vec3(       0, -10002.0,        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.

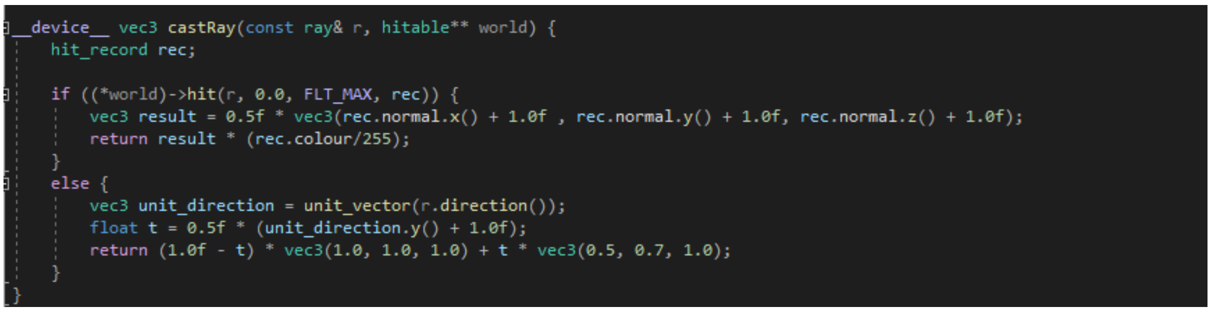
2. In order to achieve this:

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;
}
```

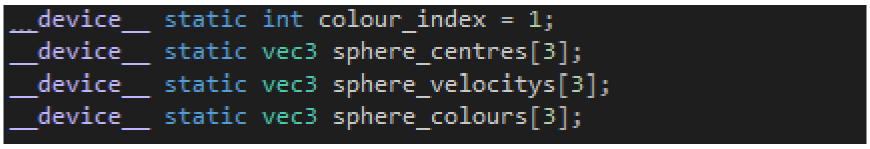
Modified the cast ray function so that when setting the colour, it applies the colour from the sphere that was hit to the shader.



```
__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);
    }
}
```

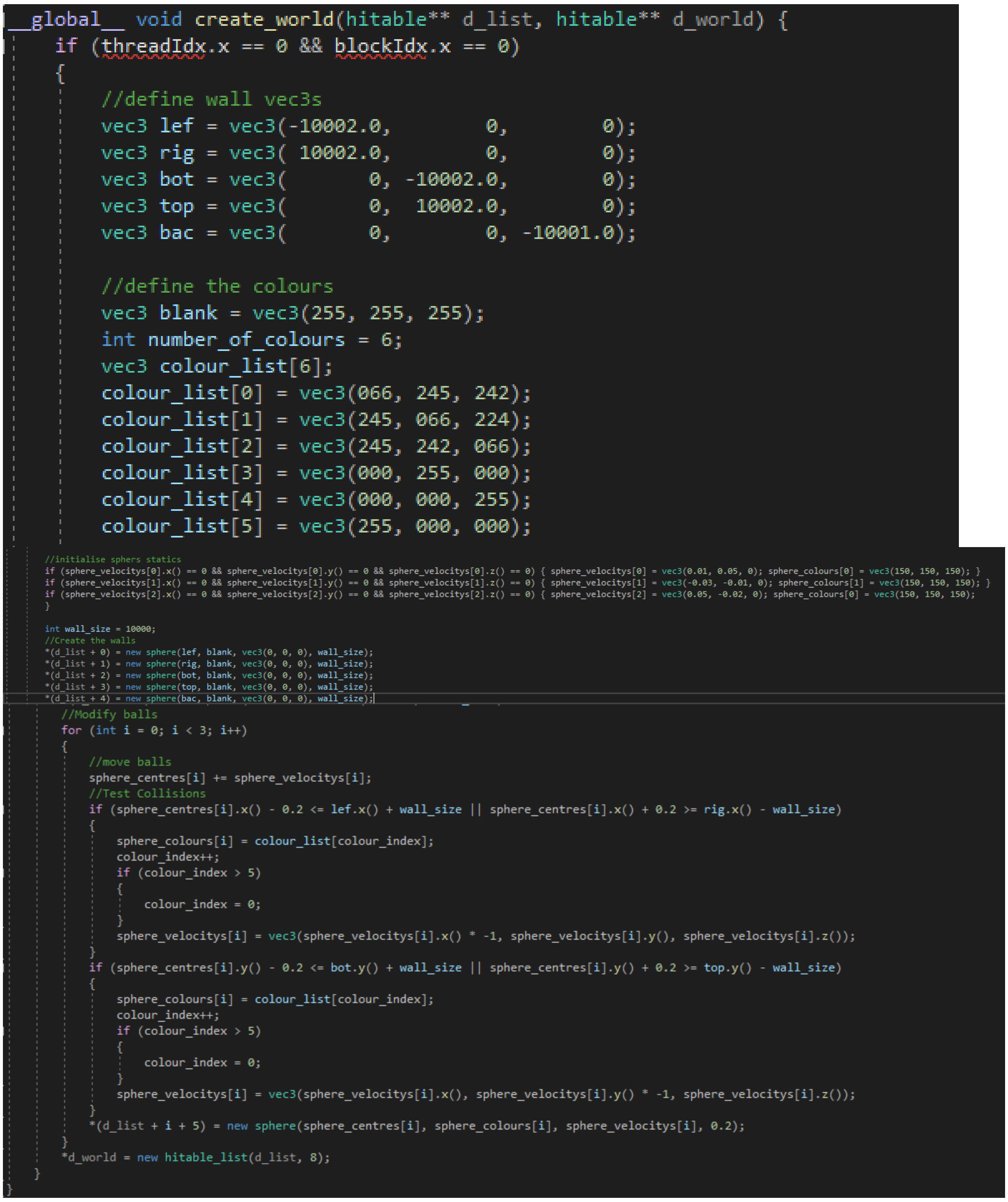
The results of this can be seen in sample output section 2

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



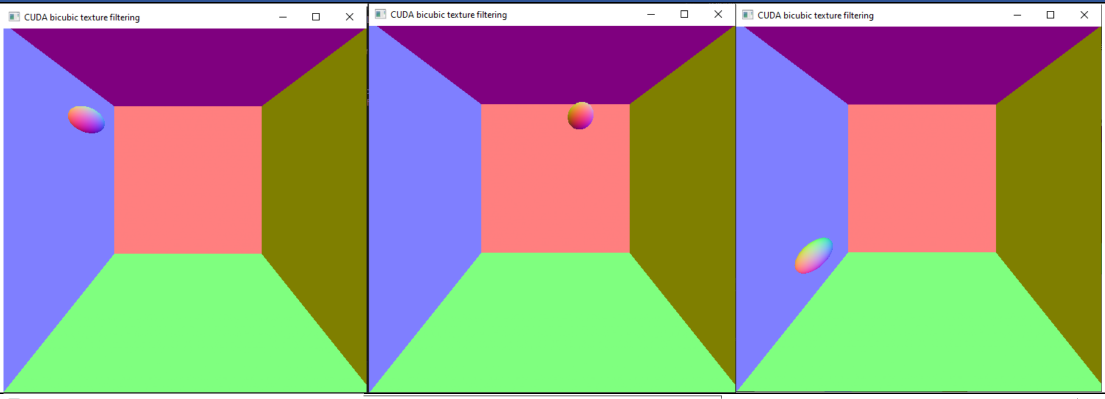
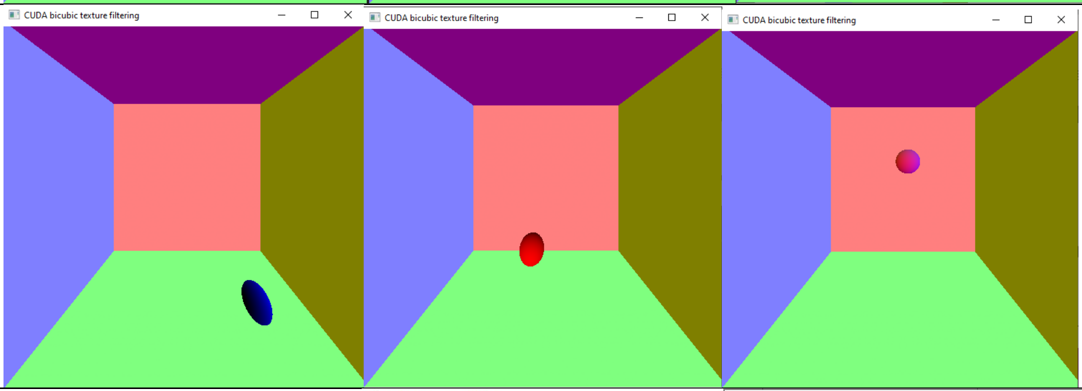
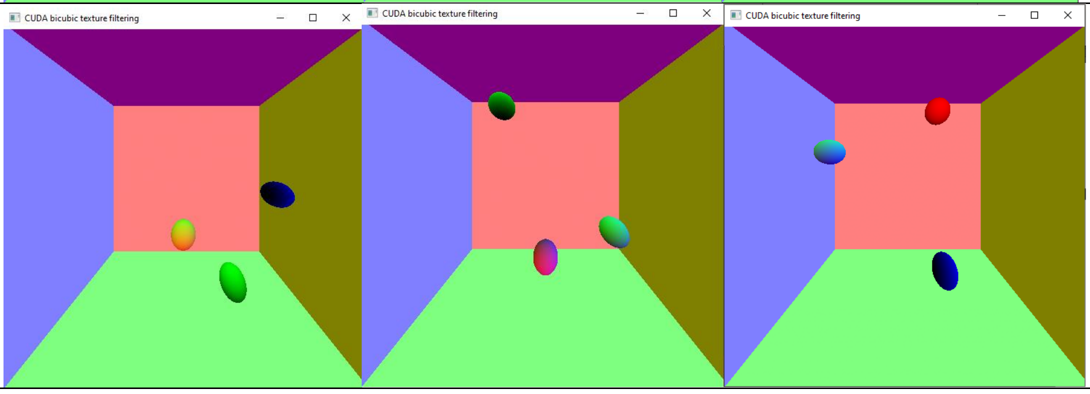
```
__global__ void create_world(hitable** d_list, hitable** d_world) {
    if (threadIdx.x == 0 && blockIdx.x == 0)
    {
        //define wall vec3s
        vec3 lef = vec3(-10002.0,        0,        0);
        vec3 rig = vec3( 10002.0,        0,        0);
        vec3 bot = vec3(       0, -10002.0,        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_velocitys[0].x() == 0 && sphere_velocitys[0].y() == 0 && sphere_velocitys[0].z() == 0) { sphere_velocitys[0] = vec3(0.01, 0.05, 0); sphere_colours[0] = vec3(150, 150, 150); }
        if (sphere_velocitys[1].x() == 0 && sphere_velocitys[1].y() == 0 && sphere_velocitys[1].z() == 0) { sphere_velocitys[1] = vec3(-0.03, -0.01, 0); sphere_colours[1] = vec3(150, 150, 150); }
        if (sphere_velocitys[2].x() == 0 && sphere_velocitys[2].y() == 0 && sphere_velocitys[2].z() == 0) { sphere_velocitys[2] = vec3(0.05, -0.02, 0); sphere_colours[0] = 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_velocitys[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_velocitys[i] = vec3(sphere_velocitys[i].x() * -1, sphere_velocitys[i].y(), sphere_velocitys[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_velocitys[i] = vec3(sphere_velocitys[i].x(), sphere_velocitys[i].y() * -1, sphere_velocitys[i].z());
            }
            *(d_list + i + 5) = new sphere(sphere_centres[i], sphere_colours[i], sphere_velocitys[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:

Ref	output
1	
2	
3	

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:

600086 Lab Book

Week 2 – Lab B

Date: 12th Feb 2022

Q1. First threads

Question:

Replace the synchronous call to your function with an asynchronous call.

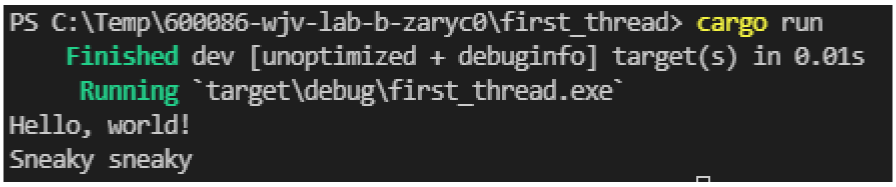
Solution:

```
fn main()
{
    std::thread::spawn(move || my_function() );
    std::thread::spawn(move || second_function() );
    std::thread::sleep(std::time::Duration::new(1,0));
}

fn my_function()
{
    println!("Hello, world!");
}

fn second_function()
{
    println!("Sneaky sneaky")
}
```

Test data:

n/a

Sample output:



```
PS C:\Temp\600086-wjv-lab-b-zaryc0\first_thread> cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target\debug\first_thread.exe`
Hello, world!
Sneaky sneaky
```

Reflection:

This is threading 101. Adding the Time delay thread to the function is necessary to alleviate the race condition created by spawning asynchronous threads with no other content to the program causing it to terminate before all of the thread shave a chance to complete.

Metadata:

Threads

Further information:

Unsure of the use of mut?

Q2. Joining threads

Question:

Replace the synchronous call to your function with an asynchronous call.

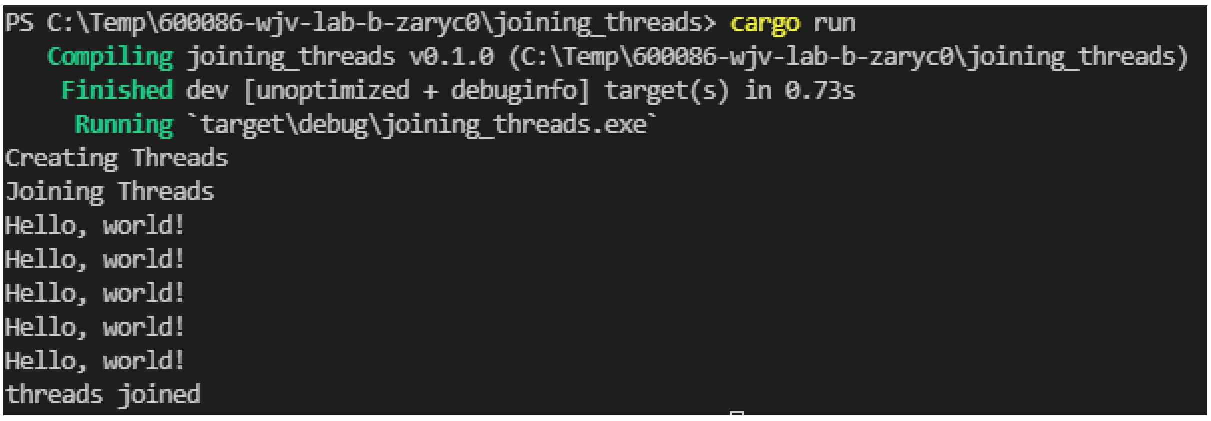
Solution:

```rust
fn main()
{
    let num_of_threads : i32 = 5;
    let mut list_of_threads = vec!();

    println!("Creating Threads");

    for _id in 0..num_of_threads
    {
        list_of_threads.push(std::thread::spawn(move || my_function() ));
    }

    println!("Joining Threads");
    for thread in list_of_threads
    {
        thread.join().expect("join failed");
    }
    println!("threads joined");
}

fn my_function()
{
    println!("Hello, world!");
}
```

Test data:

n/a

Sample output:



```
PS C:\Temp\600086-wjv-lab-b-zaryc0\joining_threads> cargo run
   Compiling joining_threads v0.1.0 (C:\Temp\600086-wjv-lab-b-zaryc0\joining_threads)
    Finished dev [unoptimized + debuginfo] target(s) in 0.73s
     Running `target\debug\joining_threads.exe`
Creating Threads
Joining Threads
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
threads joined
```

Reflection:

This is joining the threads back together again it resynchronizing them ensuring that all of the threads have completed their tasks before ending the program this helps alleviate the race condition from the Q1 solution where the threads have to race against the main program to execute their functions.

Metadata:

Threads

Further information:

Q3 Experimentation

Question:

Now that you have the basic framework for creating and joining threads, experiment with giving the threads items of work, as well as altering the number of threads used.

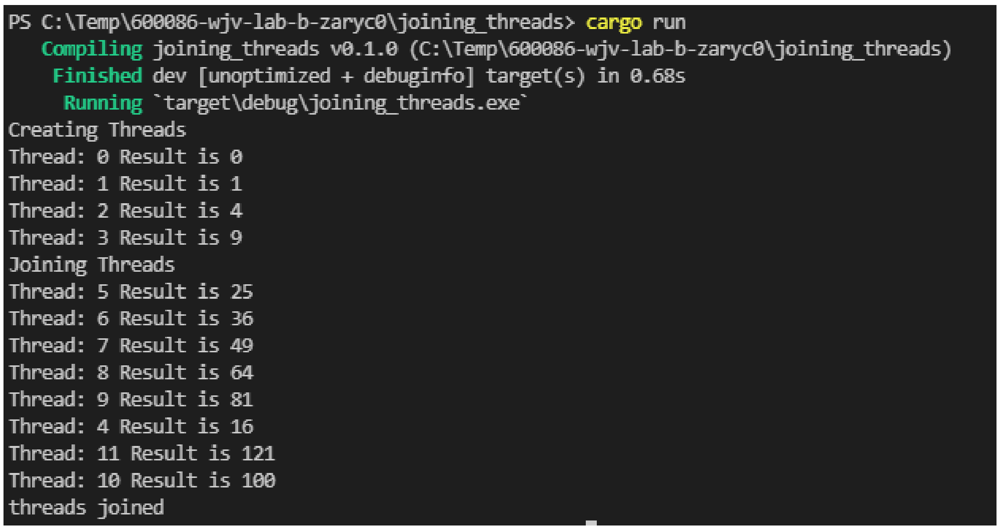
Solution:

```rust
fn main()
{
    let num_of_threads : i32 = 12;
    let mut list_of_threads = vec!();

    println!("Creating Threads");

    for _id in 0..num_of_threads
    {
        list_of_threads.push(std::thread::spawn(move || perform_task(_id) ));
    }

    println!("Joining Threads");
    for thread in list_of_threads
    {
        thread.join().expect("join failed");
    }
    println!("threads joined");
}

fn perform_task(id:i32)
{
    let result:i32 = id * id;
    println!("Thread: {} Result is {}", id,result);
}
```

Test data:

n/a

Sample output:



```
PS C:\Temp\600086-wjv-lab-b-zaryc0\joining_threads> cargo run
   Compiling joining_threads v0.1.0 (C:\Temp\600086-wjv-lab-b-zaryc0\joining_threads)
    Finished dev [unoptimized + debuginfo] target(s) in 0.68s
     Running `target\debug\joining_threads.exe`
Creating Threads
Thread: 0 Result is 0
Thread: 1 Result is 1
Thread: 2 Result is 4
Thread: 3 Result is 9
Joining Threads
Thread: 5 Result is 25
Thread: 6 Result is 36
Thread: 7 Result is 49
Thread: 8 Result is 64
Thread: 9 Result is 81
Thread: 4 Result is 16
Thread: 11 Result is 121
Thread: 10 Result is 100
threads joined
```

Reflection:
I added a task to square the ID value as part of the thread function it then printed the result to the terminal along with the thread ID.

Metadata:
Threads

Further information:

600086 Lab Book

Week 3 – Lab C

Date: 21st Feb 2022

Q1. Multiple Rust files

Question a):

Accessing a secondary rust file using the mod statement and use statements

Solution:

Main

```rust
mod my_second_file;
use my_second_file::run;
fn main() {

    run();
}
```

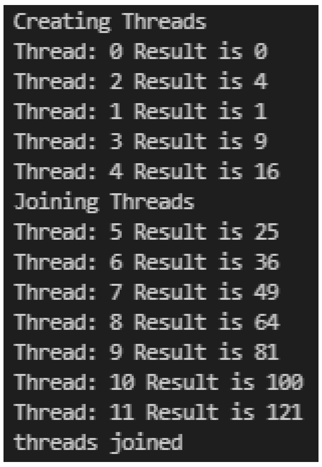
Second file

```rust
pub fn run()
{
    let num_of_threads : i32 = 12;
    let mut list_of_threads = vec!();

    println!("Creating Threads");

    for _id in 0..num_of_threads
    {
        list_of_threads.push(std::thread::spawn(move || perform_task(_id) ));
    }

    println!("Joining Threads");
    for thread in list_of_threads
    {
        thread.join().expect("join failed");
    }
    println!("threads joined");
}

fn perform_task(id:i32)
{
    let result:i32 = id * id;
    println!("Thread: {} Result is {}", id,result);
}
```

Test data:

n/a

Sample output:



```
Creating Threads
Thread: 0 Result is 0
Thread: 2 Result is 4
Thread: 1 Result is 1
Thread: 3 Result is 9
Thread: 4 Result is 16
Joining Threads
Thread: 5 Result is 25
Thread: 6 Result is 36
Thread: 7 Result is 49
Thread: 8 Result is 64
Thread: 9 Result is 81
Thread: 10 Result is 100
Thread: 11 Result is 121
threads joined
```

Reflection:

This is useful for organising code into different files based on area of concern.

Metadata:

“Multiple Rust files”,”mod”,” Ownership”,”use”

Further information:

None

Q2. Ownership

Question:

A) The main function creates a Person struct and passes it to the print_person.
Add a second call to print_person within main, to print out the details a second time. Why does this not compile?

B) Rust has a rule that states you cannot have more than one mutable reference to the same object, neither can you have even a single mutable reference to an object that has one or more immutable references. So why does the code you have created, work?

C) Rust has seen that we are trying to use a mutable reference to an object for which there is still an active immutable reference. How can this be solved?

Solution:

A)

```
fn main() {
    let p1 = Person::new("Jane", 30);

    print_person(&p1);
    print_person(&p1)
}

fn print_person(p: &Person) {
    println!("{} is {} years old", p.name, p.age);
}
```

B)

```
fn main() {
    let mut p1 = Person::new("Jane", 30);

    print_person(&p1);
    print_person(&p1);

    increment_age(&mut p1);
    increment_age(&mut p1);
}

fn print_person(p: & Person) {
    println!("{} is {} years old", p.name, p.age);
}

fn increment_age(p: &mut Person){
    p.age = p.age + 1;
}
```

C)

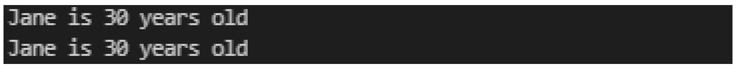
```
fn main() {
    let mut p1 = Person::new("Jane", 30);

    let r1 = &p1;

    print_person(r1);

    let r3 = &mut p1;

    increment_age(r3);
    increment_age(r3);

    let r2 = &p1;
    print_person(r2);
}

fn print_person(p: & Person) {
    println!("{} is {} years old", p.name, p.age);
}

fn increment_age(p: &mut Person){
    p.age = p.age + 1;
}
```

Test data:

n/a

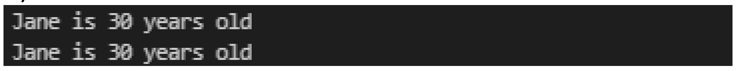
Sample output:

A)



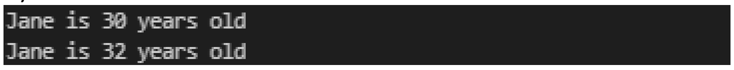
```
Jane is 30 years old
Jane is 30 years old
```

B)



```
Jane is 30 years old
Jane is 30 years old
```

C)



```
Jane is 30 years old
Jane is 32 years old
```

Reflection:

A) The code will not compile because the print_person(p1) does not use call by value and so passes ownership of the struct to the initial function call meaning no other functions can interact with it because the struct itself is passed into the function meaning that p1 is now unavailable. This can be rectified by modifying print_person(p1) to utilise a reference which will results in the p1 variable being accessible multiple times

B) The using the mutable reference should not allow the code to work however in the given circumstance the references aren't used in an order that would cause an issue and so the Rust compiler allows it to pass

C) Rust will not allow an immutable reference to be called after a mutable one if it was instantiated prior to the mutable reference as in the example code. In order to solve this instantiating, the immutable reference just prior to use allows the code to now run. This is due to the read-only reference(immutable) could become out of date if called once more after a read/Write reference(mutable) has been used.

Metadata:

"Multiple Rust files","mod"," Ownership","use"

Further information:

None

Q3. Classes

Question:

Implementing a class from a second file to allow data to be shared between threads.

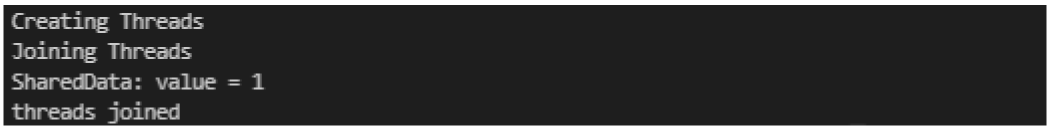
Solution:

```rust
mod shared_data;
use shared_data::SharedData;

fn main()
{
    let mut sd = SharedData::new();


    let mut list_of_threads = vec!();

    println!("Creating Threads");

    list_of_threads.push(std::thread::spawn(move || perform_task(& mut sd) ));

    println!("Joining Threads");
    for thread in list_of_threads
    {
        thread.join().expect("join failed");
    }
    println!("threads joined");
}

fn perform_task(shared : &mut SharedData)
{
    shared.update();
    shared.print();
}
```

Test data:

n/a

Sample output:



```
Creating Threads
Joining Threads
SharedData: value = 1
threads joined
```

Reflection:

This is a useful method for sharing data however I could not get it to work with multiple threads at the same time as attempting to reuse the &mut Shared Data reference causes an error on the move closure in the thread::spawn() function. And attempting to create an additional &mut reference caused a different error.

Metadata:

“Multiple Rust files”,”mod”,” Ownership”,”use”,”Classes”

Further information:

Unsure of how to allow multiple writeable references to be used. Also find the &mut to be extremely confusing.

600086 Lab Book

Week 4 – 600086-Lab-D

Date: 1st Mar 2022

Q1. Ownership limitations

Question:

Try to expand the code to include a data member in Engine that links to the Aircraft.

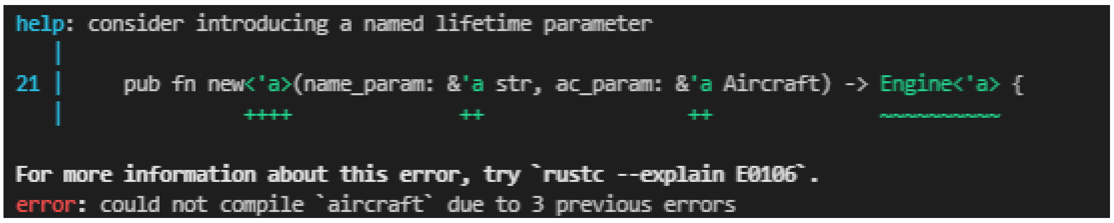
Solution:

```
struct Aircraft<'a> {
    name: String,
    engines: Vec<&'a Engine>,
}
impl Aircraft<'_> {
    pub fn new(name_param: &str) -> Aircraft {
        Aircraft {
            name: name_param.to_string(),
            engines: Vec::new()
        }
    }
}
struct Engine<'a> {
    aircraft : Aircraft<'a>,
    name: String,
}
impl Engine {
    pub fn new(name_param: &str, ac_param: &Aircraft) -> Engine {
        Engine {
            name: name_param.to_string(),
            aircraft: ac_param
        }
    }
}

fn main() {
    let engine1 = Engine::new( "General Electric F404" );
    let engine2 = Engine::new( "General Electric F404" );
    let mut f18 = Aircraft::new( "F-18" );

    f18.engines.push (&engine1);
    f18.engines.push (&engine2);

    println! ("Aircraft: {} has a {} and {} ", f18.name, f18.engines[0].name,
f18.engines[1].name );
}
```

Test data:

n/a

Sample output:



```
help: consider introducing a named lifetime parameter
   |
21 |     pub fn new<'a>(name_param: &'a str, ac_param: &'a Aircraft) -> Engine<'a> {
   |               ++++              ++                  ++               ~~~~~~~~~~

For more information about this error, try `rustc --explain E0106`.
error: could not compile `aircraft` due to 3 previous errors
```

Reflection:

This cannot be done as the engines are owned by the aircraft class therefore the engines cannot own a reference to the aircraft in the way specified as this would create a circular reference.

Metadata:

F18 , Aircraft,ownership

Further information:

Q2. Reference counters 1

Question:

Use your knowledge of the Rust ownership model to explain what is happening with the reference counters and why we do not need to pass them as references.

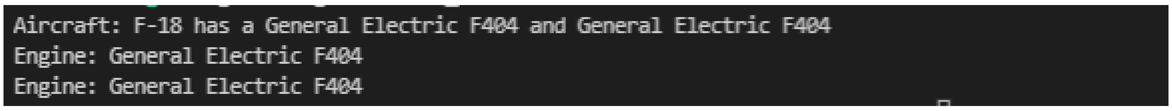
Solution:

```rust
use std::rc::Rc;

struct Aircraft {
    name: String,
    engines: Vec<Rc<Engine>>,
}

impl Aircraft {
    pub fn new(name_param: &str) -> Aircraft {
        Aircraft {
            name: name_param.to_string(),
            engines: Vec::new()
        }
    }
}

struct Engine {
    name: String,
}

impl Engine {
    pub fn new(name_param: &str) -> Engine {
        Engine {
            name: name_param.to_string(),
        }
    }
}

fn main() {
    let engine1 = Rc::new(Engine::new( "General Electric F404" ));
    let engine2 = Rc::new(Engine::new( "General Electric F404" ));

    let mut f18 = Aircraft::new( "F-18" );

    f18.engines.push (engine1.clone());
    f18.engines.push (engine2.clone());

    println! ("Aircraft: {} has a {} and {}", f18.name, f18.engines[0].name ,
f18.engines[1].name );
    println! ("Engine: {} ", engine1.name );
    println! ("Engine: {} ", engine2.name );
}
```

Test data:

n/a

Sample output:



```
Aircraft: F-18 has a General Electric F404 and General Electric F404
Engine: General Electric F404
Engine: General Electric F404
```

Reflection:

The RCs do not require passing in as references as the engines are owned by the RC the engine1 and engine2 variables are actually RC references to the memory locations where the instance of the Engine is stored so when creating the aircraft rather than moving ownership it creates a new instance that is a clone of the one owned by the RC.

Metadata:

F18 , Aircraft,ownership

Further information:

Q2. Reference counters 2

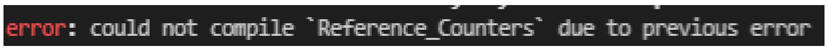
Question:

Remove the clone() method from this line
f18.engines.push (engine1.clone());
Can you explain why this program now fails to build?

Solution:

```rust
use std::rc::Rc;

struct Aircraft {
    name: String,
    engines: Vec<Rc<Engine>>,
}

impl Aircraft {
    pub fn new(name_param: &str) -> Aircraft {
        Aircraft {
            name: name_param.to_string(),
            engines: Vec::new()
        }
    }
}

struct Engine {
    name: String,
}

impl Engine {
    pub fn new(name_param: &str) -> Engine {
        Engine {
            name: name_param.to_string(),
        }
    }
}

fn main() {
    let engine1 = Rc::new(Engine::new( "General Electric F404" ));
    let engine2 = Rc::new(Engine::new( "General Electric F404" ));

    let mut f18 = Aircraft::new( "F-18" );

    f18.engines.push (engine1);
    f18.engines.push (engine2.clone());

    println! ("Aircraft: {} has a {} and {}", f18.name, f18.engines[0].name ,
f18.engines[1].name );
    println! ("Engine: {} ", engine1.name );
    println! ("Engine: {} ", engine2.name );
}
```

Test data:
n/a

Sample output:


```
error: could not compile `Reference_Counters` due to previous error
```

Reflection:
The code will no longer compile because unlike before where the instanced engine object ref is cloned into a new ref of engine when passed in, the code is now attempting to pass an object that is of type RC<Engine> which is not an engine but is actually the memory location where the RC is storing the engine object and therefore lacks the required properties that are called for in the programme e.g name.

Metadata:
F18 , Aircraft,ownership

Further information:

Q2. Reference counters 3

Question:

You should be able to service engine3.
You will get a build error if you try and service engine2, which is accessed through an rc.
Again, using your knowledge of the Rust ownership module can you explain why the error is occurring?

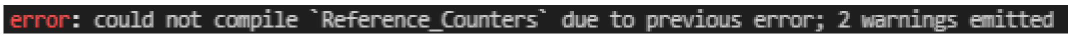
Solution:

```rust
use std::rc::Rc;
struct Aircraft {
    name: String,
    engines: Vec<Rc<Engine>>,
}
impl Aircraft {
    pub fn new(name_param: &str) -> Aircraft {
        Aircraft {
            name: name_param.to_string(),
            engines: Vec::new()
        }
    }
}
struct Engine {
    name: String,
    requires_service: bool,
}
impl Engine {
    pub fn new(name_param: &str) -> Engine {
        Engine {
            name: name_param.to_string(),
            requires_service: false,
        }
    }
    pub fn service(&mut self) {
        self.requires_service = false;
    }
}
fn main() {
    let engine1 = Rc::new(Engine::new( "General Electric F404" ));
    let engine2 = Rc::new(Engine::new( "General Electric F404" ));
    let mut engine3 = Engine::new( "General Electric F404" );
    engine2.service();
    let mut f18 = Aircraft::new( "F-18" );
    f18.engines.push (engine1.clone());
    f18.engines.push (engine2.clone());

    println! ("Aircraft: {} has a {} and {}", f18.name, f18.engines[0].name , f18.engines[1].name );
    println! ("Engine: {} ", engine1.name );
    println! ("Engine: {} ", engine2.name );
}
```

Test data:

n/a

Sample output:



```
error: could not compile `Reference_Counters` due to previous error; 2 warnings emitted
```

Reflection:

The Program will not build when attempting to call service() on engine2 as the reference owned by the RC is immutable and so cannot be borrowed as a mutable reference, this is an issue as the service() requires a mutable reference as it writes to the properties of the engine.

Metadata:

F18 , Aircraft,ownership

Further information:

600086 Lab Book

Week 5 – Lab E

Date: 9th Mar 2022

Q1. Thread safe printing

Question:

Build and run the unsafe_print folder

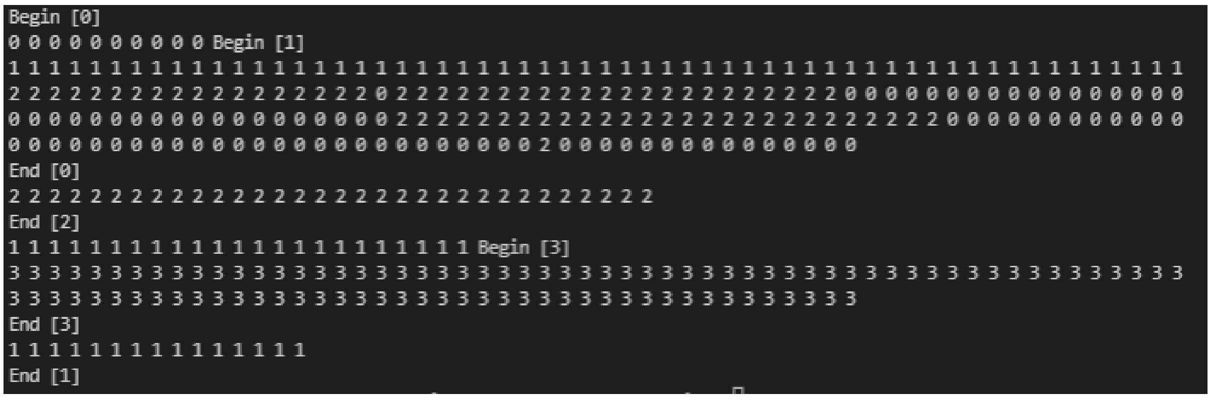
Solution:

```rust
fn main() {
    let num_of_threads = 4;
    let mut array_of_threads = vec!();

    for id in 0..num_of_threads {
        array_of_threads.push(std::thread::spawn(move || print_lots(id)) );
    }

    for t in array_of_threads {
        t.join().expect("Thread join failure");
    }
}

fn print_lots(id: u32) {
    println!("Begin [{}]", id);
    for _i in 0..100 {
        print!("{} ", id);
    }
    println!("\nEnd [{}]", id);
}
```

Test data:

N/A

Sample output:



```
Begin [0]
0 0 0 0 0 0 0 0 0 0 Begin [1]
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
End [0]
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
End [2]
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 Begin [3]
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
End [3]
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
End [1]
```

Reflection:
The prints appear out of order and seemingly random as each thread is executing entirely independently this means that there is nothing to control the order and ensure that they are printed in sync. This is known as a race condition as each thread is effectively racing to print first.

Metadata:
Race Conditions Unsafe Print

Further information:
none

Question:

Reproduce the unsafe_print code accounting for the race condition

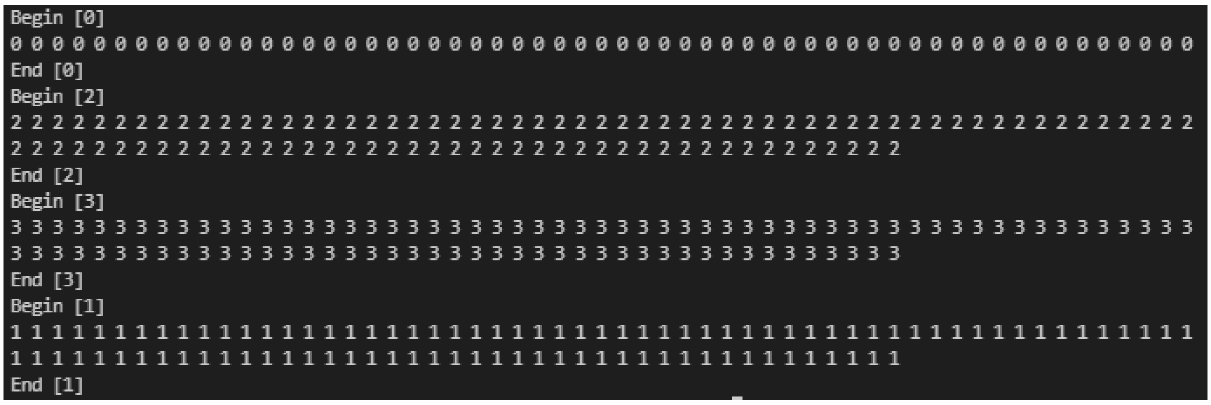
Solution:

```rust
use std::sync::{Arc, Mutex};

fn main() {
    let num_of_threads = 4;
    let mut array_of_threads = vec!();
    let counter = Arc::new(Mutex::new(0));

    for id in 0..num_of_threads {
        let counter_clone = counter.clone();
        array_of_threads.push(std::thread::spawn(move ||
print_lots(id,counter_clone)) );
    }

    for t in array_of_threads {
        t.join().expect("Thread join failure");
    }
}

fn print_lots(id: u32, c:Arc<Mutex<u32>>) {

    let _guard = c.lock().unwrap();
    println!("Begin [{}]", id);
    for _i in 0..100 {
        print!("{} ", id);
    }
    println!("\nEnd [{}]", id);
}
```

Test data:

N/A

Sample output:



```
Begin [0]
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
End [0]
Begin [2]
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
End [2]
Begin [3]
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
End [3]
Begin [1]
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
End [1]
```

Reflection:

The output is still not in order but each section is uninterrupted. To achieve this I created a counter Arc<Mutex<u32>> and passed this into the thread when a thread entered the function it locked the. Arc<Mutex<u32>> which was a shared asynchronous reference this meant that the other threads could not continue to the print function until the current thread released the lock when it fell out of scope.

What happens to your code if you fail to release the mutex?

Should the mutex fail to be released then the code will not complete as the remaining threads would be locked out of completing indefinitely.

What happens if you raise an exception within the critical section?

Should the print statement return an error or throw an exception for any reason then the function would be left without leaving scope keeping the lock in place.

There is also still a race condition albeit a slightly less crucial one as the threads are now effectively racing to be the thread to lock the Arc<Mutex<u32>> and as such the order of prints is not guaranteed.

Metadata:

Safe print ARC Mutex Lock

Further information:

How do you raise an exception that can cause the thread to end silently so as to prove the principles stated above?

Q2. Triangles using OpenGL

Question:

Adapt the provided code to produce a more chaotic movement of triangles.

Solution:

```
    let mut delta_x: f32 = -0.5;
    let mut delta_y: f32 = -0.5;
    let mut randomizer= rand::thread_rng();
//
…
// Begin render loop

        // Animation counter
        let x:f32 = randomizer.gen_range(-0.02..0.02);
        let y:f32 = randomizer.gen_range(-0.02..0.02);
        delta_x += x;
        if delta_x > 0.7 {
            delta_x = -1.4;
        }
        if delta_x < -1.4{
            delta_x = 0.7
        }
        delta_y += y;
        if delta_y > 0.7 {
            delta_y = -1.4;
        }
        if delta_y < -1.4{
            delta_y = 0.7
        }

        // Create a drawing target
        let mut target = display.draw();

        // Clear the screen to black
        target.clear_color(0.0, 0.0, 0.0, 1.0);

        // Iterate over the 10 triangles
        for i in 0 .. 10 {

            // Calculate the position of the triangle
            let pos_x : f32 = delta_x + ((i as f32) * 0.1);
            let pos_y : f32 = delta_y + ((i as f32) * 0.1);
            let pos_z : f32 = 0.0;

            // Create a 4x4 matrix to store the position and orientation of
the triangle
            let uniforms = uniform! {
                matrix: [
                    [1.0, 0.0, 0.0, 0.0],
                    [0.0, 1.0, 0.0, 0.0],
                    [0.0, 0.0, 1.0, 0.0],
                    [pos_x, pos_y, pos_z, 1.0],
                ]
            };

            // Draw the triangle
            target.draw(&vertex_buffer, &indices, &program, &uniforms,
&Default::default()).unwrap();
        }

        // Display the completed drawing
        target.finish().unwrap();

        // End render loop
```

Test data:
N/A

Sample output:
Can't be shown as it is video but will present in Lab.

Reflection:
Modified delta to be replaced by two variables one governing the X dimension increment and the other governing the y dimension increment:

delta_x

delta_y

I also introduced a random variable that can be used to generate a random number in a range as done at the start of the //Animation Counter section. I also allowed the triangles to move in all directions by introducing negative increment values into the range. Because I had now implemented negative movement, I had to expand the value checks to allow for the reverse case otherwise the triangles can travel backwards indefinitely.

Metadata:
OpenGL

Further information:
N/A

600086 Lab Book

Week 6 – Lab F

Date: 16th Mar 2022

Q1. Particles

Question: Create a particle system that can manage moving particles

Solution:

Opened the empty particles project and created the Particle and Particle system structs these are as follows.

```rust
#[derive(Debug, Copy, Clone)]
struct Particle {
    x_dir : f32,
    y_dir : f32,
    x_pos : f32,
    y_pos : f32,
    velocity: f32
}
impl Particle {
    pub fn new( xd : f32, yd :f32, x:f32 , y: f32, v:f32) -> Particle {
        Particle{
            x_dir : xd,
            y_dir : yd,
            x_pos : x,
            y_pos : y,
            velocity : v,
        }
    }
    pub fn move_particle(&mut self) {
        let x2 = self.x_dir * self.velocity;
        let y2 = self.y_dir * self.velocity;
        let test = self.x_pos + x2;
        if test < 0.0 || test > 10.0 {
            self.x_dir *= -1.0;
        }
        let test = self.y_pos + y2;
        if test < 0.0 || test > 10.0 {
            self.y_dir *= -1.0;
        }
        self.x_pos += x2;
        self.y_pos += y2;
    }
}
```

```
struct ParticleSystem
{
    particles: Vec<Particle>,
}
impl ParticleSystem
{
    pub fn new() -> ParticleSystem
    {
        ParticleSystem
        {
            particles: Vec::new(),
        }
    }
    pub fn run_system(&mut self)
    {
        for particle in self.particles.iter_mut()
        {
            let p = particle;
            p.move_particle();
        }
    }
}
```

I then implemented the open GL code from last week in order to display the particles and show the movement. The implementation can be seen below, note only the render loop is shown

```
// Begin render loop

        // Animation counter

        delta_t += 0.005;
        if delta_t > 0.7 {
            delta_t = -1.4;
        }

        // Create a drawing target
        let mut target = display.draw();

        // Clear the screen to black
        target.clear_color(0.0, 0.0, 0.0, 1.0);

        //run particle system
        if T.elapsed().as_secs()<10
        {
            p.run_system();
        }
        // Iterate over the 10 triangles
        let particle_list = &p.particles;
        let mut i = 0;
        for particle in particle_list
        {
            // Calculate the position of the triangle
            let pos_x : f32 = (particle.x_pos -
(WINDOW_WIDTH/2.0))/(WINDOW_WIDTH/2.0);
            let pos_y : f32 = (particle.y_pos -
(WINDOW_HEIGHT/2.0))/(WINDOW_HEIGHT/2.0);
            let pos_z : f32 = 0.0;

            //calculate colors
            let mut r = rand::thread_rng();
            let red   : f32 = r.gen_range(0.0..1.0);
            let green: f32 = r.gen_range(0.0..1.0);
            let blue : f32 = r.gen_range(0.0..1.0);

            // Create a 4x4 matrix to store the position and orientation of
the triangle
            let uniforms = uniform! {
                matrix: [
                    [1.0, 0.0, 0.0, 0.0],
                    [0.0, 1.0, 0.0, 0.0],
                    [0.0, 0.0, 1.0, 0.0],
                    [pos_x, pos_y, pos_z, 1.0],
                ],
                color: [red,green,blue,1.0 as f32]
            };
            i += 1;

            // Draw the triangle
            target.draw(&vertex_buffer, &indices, &program, &uniforms,
&Default::default()).unwrap();
        }

        // Display the completed drawing
        target.finish().unwrap();

        // End render loop
```

The position of the particles was kept within a 10 by 10 enclosure defined using the following global variables

```
const WINDOW_HEIGHT: f32 = 10.0;
const PARTICLE_SPEED:f32 = 0.04;
const WINDOW_WIDTH: f32 = 10.0;
const LIMIT :u32 = 100;
```

this is then scaled to be represented on the screen as can be seen in the pos_x variable set in the previous code.

Test data:

N/A

Sample output:
Can’t be shown as it is video but will present in Lab.

Reflection:
Was fairly difficult handling the various mut, &mut, refs for the self variable. I kept getting ownership issues until I came across iter_mut() function for the VEWc<Particle>

Metadata:

Further information:
N/A

Q2. Threaded Particles

Question:

Adapt the code from Q1 to utilise multiple scoped threads for calculating the particle movement

Solution:

Added and adapted the following functions to the code in order to run the movement via scoped threads.

```
    let mut pool = scoped_threadpool::Pool::new(NUM_OF_THREADS as u32);
```

…

```
// Limit the scope of the reads to this section of code
        pool.scoped(|scope| {
            for slice in p.particles.chunks_mut(PARTICLES_PER_THREAD) {
                scope.execute(move || thread_main(slice,10.0));
            }
        });
        // Implicit join here, where all threads go out of scope.
```

…

```
fn thread_main (list: &mut [Particle], enclosure_size: f32)
{
    for particle in list.iter_mut()
    {
        let p = particle;
        p.move_particle();
    }
}
```

This code splits the list of particles into slices with each slice being managed by a different thread.

Test data:

Sample output:

Can't be shown as it is video but will present in Lab.

Reflection:

This was a lot easier to implement then the part 1 was however there was some confusion about the variable names given in the example code such as the chunk size parameter of the chunks_mut() function being a variable named NUMBER_OF_CHUNKS etc.

Between the two versions the running is barely noticeable between the two the threaded version seems fractionally faster but it is hard to tell.

Metadata:

OpenGL

Further information:

N/A

600086 Lab Book

Week 6 – Lab F

Date: 16th Mar 2022

Q1. Colliding Particles

Questions:

a) Modify the particle object so that it can collide with other particles
b) Add a scoped thread pool that handles the collisions of particles and counts them using a local variable

Solution:

a) Modified the particle struct top contain a collision function that takes another particle as a parameter

```
impl Particle
{
    pub fn new( _id:u32, xdir : f32, ydir :f32, x:f32 , y: f32,
vel:f32,radius:f32) -> Particle
    {
        Particle
        {
            id        : _id,
            x_dir     : xdir,
            y_dir     : ydir,
            x_pos     : x,
            y_pos     : y,
            velocity : vel,
            size      : radius,
        }
    }
    pub fn move_particle(&mut self, enclosure_width :f32,
enclosure_height:f32)
    {
        let x2 = self.x_dir * self.velocity;
        let y2 = self.y_dir * self.velocity;
        let test = self.x_pos + x2;
        if test < 0.0 || test > enclosure_width
        {
            self.x_dir *= -1.0;
        }

        let test = self.y_pos + y2;
        if test < 0.0 || test > enclosure_height
        {
            self.y_dir *= -1.0;
        }
        self.x_pos += x2;
        self.y_pos += y2;
    }
    pub fn collide(&mut self, p : &Particle) -> bool
    {
        let p2 = p;
        if self.id == p2.id
        {
            return false;
        }
        let delta_x = self.x_pos - p2.x_pos;
        let delta_y = self.y_pos - p2.y_pos;
        let delta_h_between_centers =f32::sqrt(f32::powf(delta_x,2.0) +
f32::powf(delta_y,2.0));
        let delta_h = delta_h_between_centers - self.size - p2.size;
        if delta_h < 0.0
        {
            return true;
        }
        return false;
    }
```

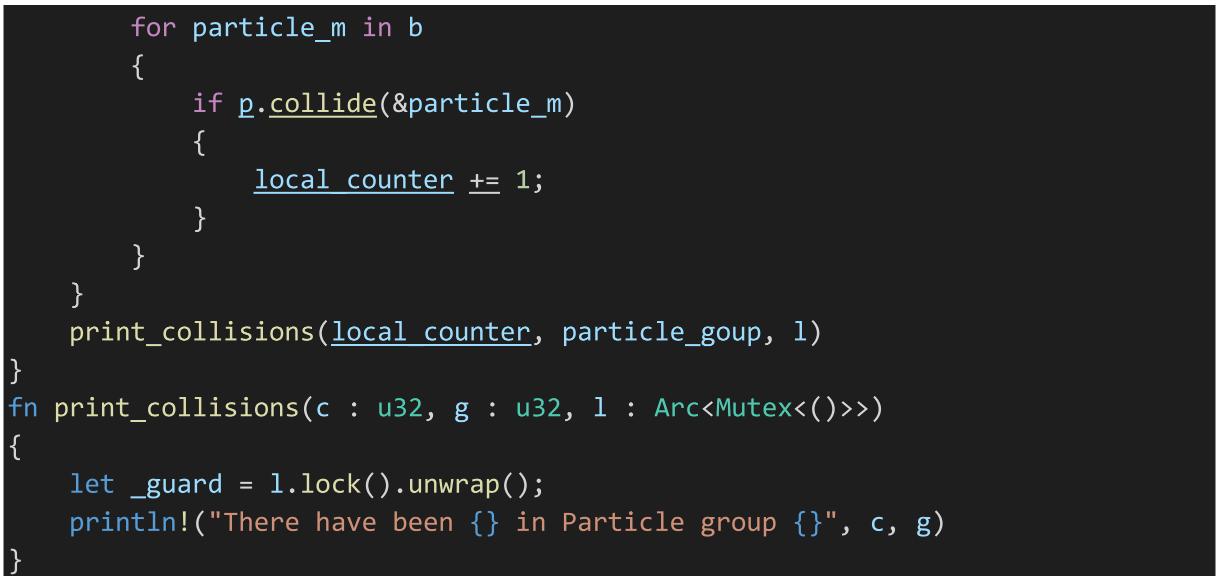
It then calculates the distance between the centres of each particle and subtracts the radius of each from the distance if the resulting value is less then 0 then the particles would have collided it then returns true, otherwise it returns false, I also added an id attribute to the particle class so that the particles can identify itself and ignore the collision check in this circumstance.

b) Added a second thread main function that takes the chunk of particles to check and a master list of particles to check against and uses a local counter to keep track

```
pool.scoped(|scope| {
    let mut i = 1;
    for slice in ref p.chunks_mut(PARTICLES_PER_THREAD) {
        let m = Clone::clone(&master_list);
        let l = global_lock.clone();
         scope.execute(move || collision_thread_main(i,slice,&m,l));
         i += 1;
    }
});
```

…

```
fn collision_thread_main(particle_goup : u32, list: &mut [Particle],
master_list:&Vec<Particle>, l : Arc<Mutex<()>>)
{
    let b = master_list;
    let a = list.iter_mut();
    let mut local_counter : u32 = 0;
    for particle_a in a
    {
        let p = particle_a;
```



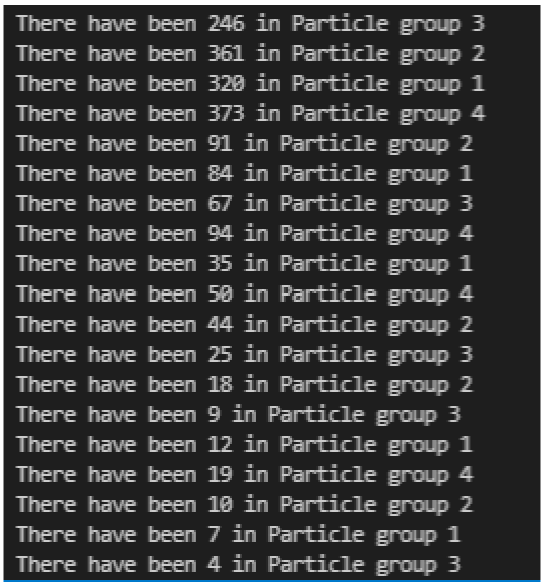
```rust
        for particle_m in b
        {
            if p.collide(&particle_m)
            {
                local_counter += 1;
            }
        }
    }
    print_collisions(local_counter, particle_goup, l)
}
fn print_collisions(c : u32, g : u32, l : Arc<Mutex<()>>)
{
    let _guard = l.lock().unwrap();
    println!("There have been {} in Particle group {}", c, g)
}
```

I also added a print function that will print the local thread count for each chunk , I implemented a guard to avoid a race condition.

Test data:

N/A

Sample output:



```
There have been 246 in Particle group 3
There have been 361 in Particle group 2
There have been 320 in Particle group 1
There have been 373 in Particle group 4
There have been 91 in Particle group 2
There have been 84 in Particle group 1
There have been 67 in Particle group 3
There have been 94 in Particle group 4
There have been 35 in Particle group 1
There have been 50 in Particle group 4
There have been 44 in Particle group 2
There have been 25 in Particle group 3
There have been 18 in Particle group 2
There have been 9 in Particle group 3
There have been 12 in Particle group 1
There have been 19 in Particle group 4
There have been 10 in Particle group 2
There have been 7 in Particle group 1
There have been 4 in Particle group 3
```

Reflection:

Had more difficulty implementing the local counter than the atomic counter.

Metadata:

Further information:

N/A

Q2. Recording collisions using an Atomic

Question:

Replace the local counter with an atomic counter to measure the number of collisions across all threads. This counter should be stored only once in the Particle System class.

Solution:

Modified the particle system class to include the Atomic reference counter

```
struct ParticleSystem
{
    collision_counter: Arc<Mutex<u32>>,
    particles: Vec<Particle>,
}
impl ParticleSystem
{
    pub fn new() -> ParticleSystem
    {
        ParticleSystem
        {
            collision_counter : Arc::new(Mutex::new(0)),
            particles: Vec::new(),
        }
    }
    pub fn run_system(&mut self,enclosure_width :f32, enclosure_height:f32)
    {
        for particle in self.particles.iter_mut()
        {
            let x = enclosure_width;
            let y = enclosure_height;
            let p = particle;
            p.move_particle(x,y);
        }
    }
}
```

This is then passed into the scoped thread main for the collision checker

```
        let master_list = Clone::clone(&p.particles);
        ref p = & mut p; // p represents my particle system
 pool.scoped(|scope| {
            for slice in ref p.particles.chunks_mut(PARTICLES_PER_THREAD) {
                let m = Clone::clone(&master_list);
                let l = ref p.collision_counter.clone();
                scope.execute(move || collision_thread_main(slice,&m,l));
            }
        });
```

The collision thread main has been modified to the following

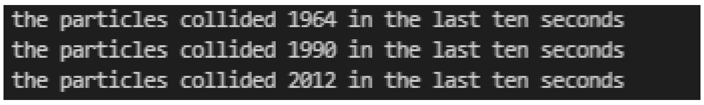
```rust
fn collision_thread_main(list: &mut [Particle], master_list:&Vec<Particle>,
atomic_counter : Arc<Mutex<u32>>)
{
    let b = master_list;
    let a = list.iter_mut();

    for particle_a in a
    {
        let p = particle_a;
        for particle_m in b
        {
            if p.collide(&particle_m)
            {
                *atomic_counter.lock().unwrap() += 1;
            }
        }
    }
}
```

The atomic_counter locks while it is updated then the next thread can also update the same value I have set the code up to print the running total of collisions every 10 seconds.

Test data:

N/A

Sample output:



```
the particles collided 1964 in the last ten seconds
the particles collided 1990 in the last ten seconds
the particles collided 2012 in the last ten seconds
```

Reflection:

The particle collisions decrease exponentially at first this is likely due to the blooming of the p[articles as they all begin in the same location and bloom out in all direction as they spread the collisions become less and less frequent.

Metadata:

OpenGL

Further information:

N/A

600086 Lab Book

Week 6 – Lab H

Date: 24th Mar 2022

Exercise 1. Condition variables

Questions:

1. create a rust program using a producer consumer architecture with condition variables to prevent race conditions
2. modify the code to allow the code to work with multiple consumers

Solution:

1.

```
use std::sync::{Arc, Mutex, Condvar};
pub fn main()
{
    println!("Begin");
    let D = Arc::new(Data::new());

    let D_clone = D.clone();

    let loops = 5;

    let producers = std::thread::spawn(move || producer_main( &D,loops));
    let consumers = std::thread::spawn(move || consumer_main(&D_clone,
loops));

    producers.join();
    consumers.join();

    println!("Cease");

}

pub fn producer_main(data : & Data, loop_limit : u32)
{
    let data = data;
    for _i in 0..loop_limit
    {
        let mut full = data.value.lock().unwrap();
        while *full
        {
            full = data.condition.wait(full).unwrap();
        }
        //"produce" by setting the mutex value to true
        *full = true;
        println!("Producer_ID: {} has produced",_i);
        data.condition.notify_all(); // notify any waiting threads
    }
}

pub fn consumer_main(data : & Data,loop_limit : u32)
{
    for _i in 0..loop_limit
    {
        let mut full = data.value.lock().unwrap();
        while !*full // de reference the guard variable
        {
            full = data.condition.wait(full).unwrap();
        }
        *full = false;
        println!("Consumer_ID: {} has consumed",_i);
        data.condition.notify_all();
    }
}

pub struct Data
{
    condition : Condvar,
    value : Mutex<bool>
}
impl Data
{
    pub fn new() -> Data
    {
        Data
        {
            condition : Condvar::new(),
            value : Mutex::new(false)
        }
    }
}
```

The above code implements a struct called Data which has a CondVal and a mutex Boolean value the Mutex bool is the value of the data and the CondVal is there to organise the access to the variable

When run it produces the output shown in the sample output section ref 1

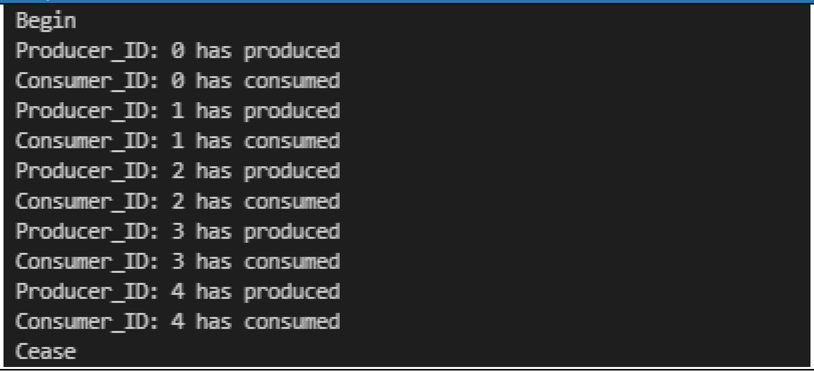
2. To run the code with multiple consumers and producers I created a thread_pool variable and pushed multiple threads of each kind to create multiple consumer and producer threads.

```rust
pub fn main()
{
    println!("Begin");
    let D = Arc::new(Data::new());
    let mut thread_pool = Vec::new();

    let loops = 5;
    let num_producers = 2;
    let num_consumers = 2;

    for _i in 0..num_producers
    {
        let d_clone = D.clone();
        thread_pool.push(std::thread::spawn(move || producer_main(&d_clone, 
loops)));
    }
    for _i in 0..num_consumers
    {
        let d_clone = D.clone();
        thread_pool.push(std::thread::spawn(move || consumer_main(&d_clone, 
loops)));
    }
    for t in thread_pool
    {
        t.join();
    }

    println!("Cease");

}
```

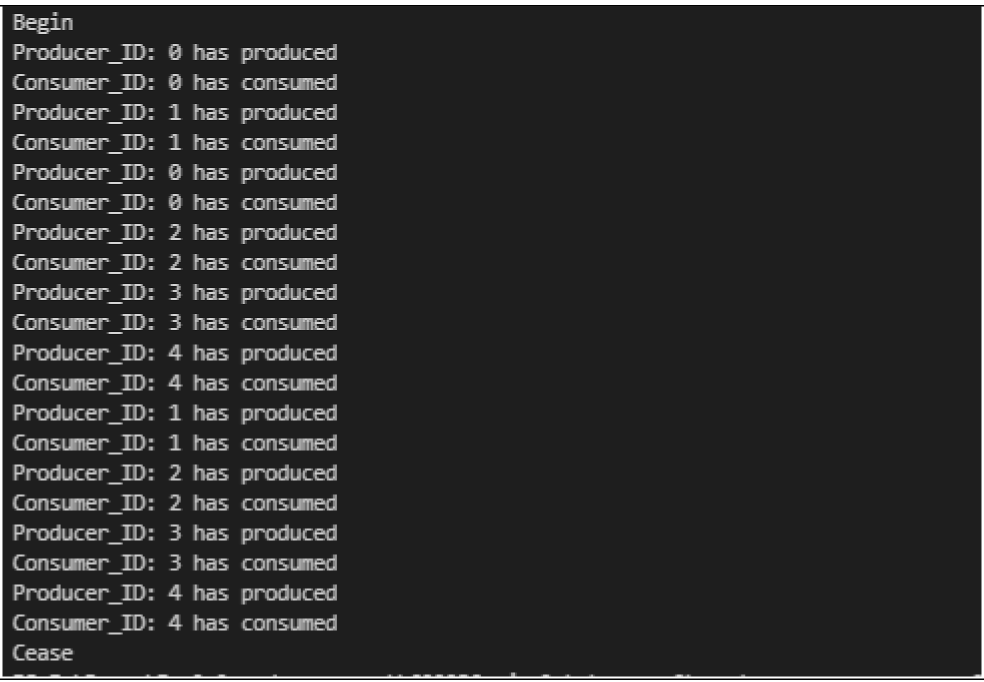
Test data:

N/A

Sample output:

ref	output
1	

```
Begin
Producer_ID: 0 has produced
Consumer_ID: 0 has consumed
Producer_ID: 1 has produced
Consumer_ID: 1 has consumed
Producer_ID: 2 has produced
Consumer_ID: 2 has consumed
Producer_ID: 3 has produced
Consumer_ID: 3 has consumed
Producer_ID: 4 has produced
Consumer_ID: 4 has consumed
Cease
```

2	

```
Begin
Producer_ID: 0 has produced
Consumer_ID: 0 has consumed
Producer_ID: 1 has produced
Consumer_ID: 1 has consumed
Producer_ID: 0 has produced
Consumer_ID: 0 has consumed
Producer_ID: 2 has produced
Consumer_ID: 2 has consumed
Producer_ID: 3 has produced
Consumer_ID: 3 has consumed
Producer_ID: 4 has produced
Consumer_ID: 4 has consumed
Producer_ID: 1 has produced
Consumer_ID: 1 has consumed
Producer_ID: 2 has produced
Consumer_ID: 2 has consumed
Producer_ID: 3 has produced
Consumer_ID: 3 has consumed
Producer_ID: 4 has produced
Consumer_ID: 4 has consumed
Cease
```

Reflection:

Part one I produced a producer consumer structured program, part 2 I scaled this program to work with multiple consumers and producers however this is not a massively scalable solution as there is still only one piece of data being accessed meaning it bottlenecks the parallel program and makes it very concurrent slightly defeating the purpose.

Metadata:

Produce, consumer, architecture, Condvar

Further information:

N/A

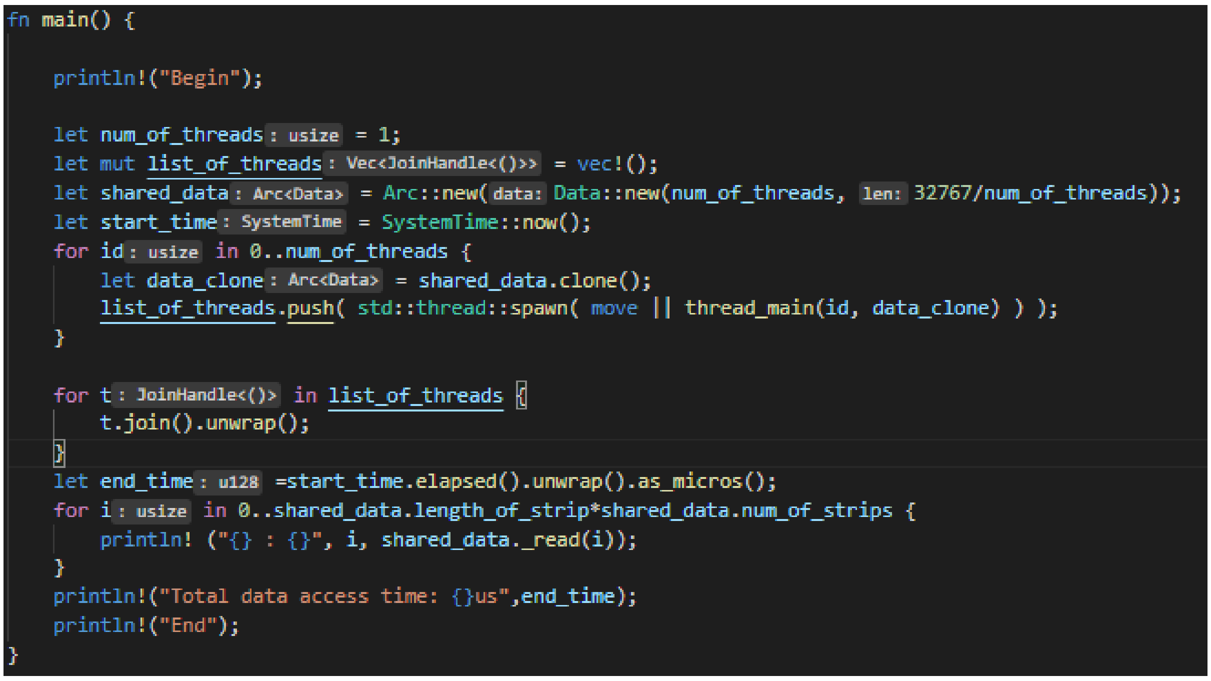
Q2. Striped Arrays

Question:

1. Add timing code to the provided program to measure the data access time, vary the number of threads to check how this changes the value
2. Modify the code to simulate random access

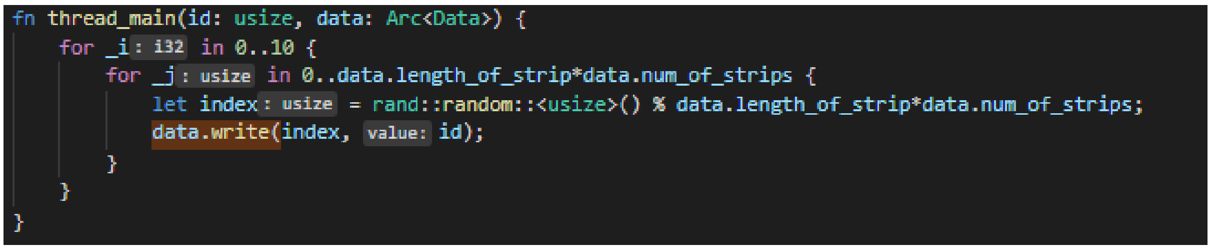
Solution:

1. Added a start_time and end_time variable to the code that wraps the thread creation and joining to time the data access portion of the programme I also modified the strip sizes so that it is the total divided by the number of threads so that the same number of accesses are made.



```rust
fn main() {

    println!("Begin");

    let num_of_threads: usize = 1;
    let mut list_of_threads: Vec<JoinHandle<()>> = vec!();
    let shared_data: Arc<Data> = Arc::new(data: Data::new(num_of_threads, len: 32767/num_of_threads));
    let start_time: SystemTime = SystemTime::now();
    for id: usize in 0..num_of_threads {
        let data_clone: Arc<Data> = shared_data.clone();
        list_of_threads.push( std::thread::spawn( move || thread_main(id, data_clone) ) );
    }

    for t: JoinHandle<()> in list_of_threads {
        t.join().unwrap();
    }
    let end_time: u128 =start_time.elapsed().unwrap().as_micros();
    for i: usize in 0..shared_data.length_of_strip*shared_data.num_of_strips {
        println! ("{} : {}", i, shared_data._read(i));
    }
    println!("Total data access time: {}us",end_time);
    println!("End");
}
```

2. To simulate random access I implemented the following code as per the lab notes in the thread_main

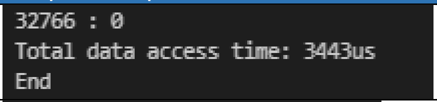
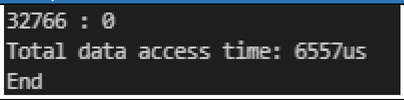
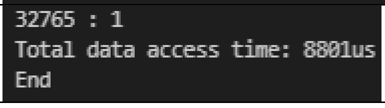
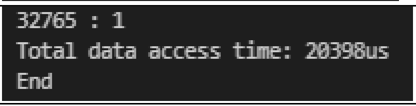


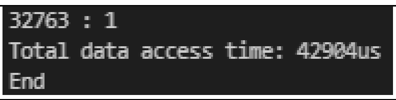
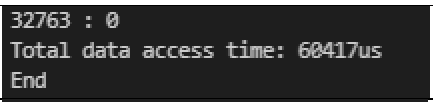
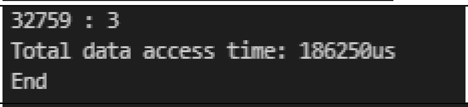
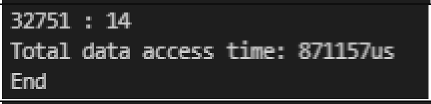
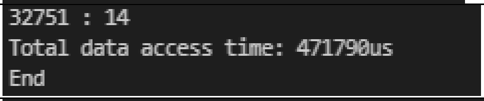
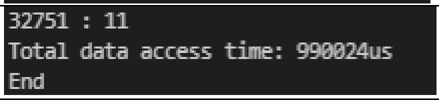
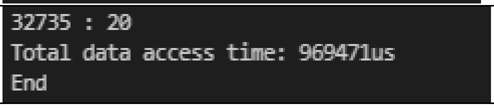
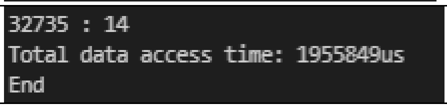
```rust
fn thread_main(id: usize, data: Arc<Data>) {
    for _i: i32 in 0..10 {
        for _j: usize in 0..data.length_of_strip*data.num_of_strips {
            let index: usize = rand::random::<usize>() % data.length_of_strip*data.num_of_strips;
            data.write(index, value: id);
        }
    }
}
```

Test data:

N/A

Sample output:

No. of threads	Output - seq	Output - rand
1	 32766 : 0 Total data access time: 3443us End	 32766 : 0 Total data access time: 6557us End
2	 32765 : 1 Total data access time: 8801us End	 32765 : 1 Total data access time: 20398us End

4	 32763 : 1 Total data access time: 42904us End	 32763 : 0 Total data access time: 60417us End
8	 32759 : 3 Total data access time: 186250us End	 32751 : 14 Total data access time: 871157us End
16	 32751 : 14 Total data access time: 471790us End	 32751 : 11 Total data access time: 990024us End
32	 32735 : 20 Total data access time: 969471us End	 32735 : 14 Total data access time: 1955849us End

Reflection:

In the sequential implementation the run time gets greater every time as the amount of work being done is very small compared to the overhead of spinning up and joining threads.

When implementing the random-access code. Expected that the code would have roughly the same access time however on average the access time was roughly double that of the sequential implementation

Metadata:

Stripped Array

Further information:

N/A