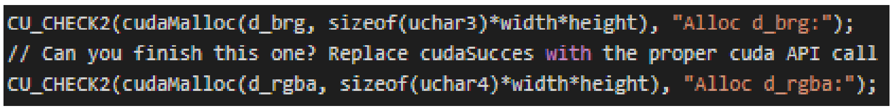
Lab 2: Cuda

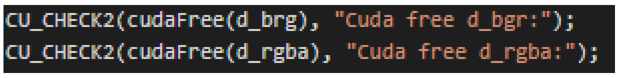
Section 1: Complete the code and execute.

First of all, we need to complete the allocation functions. The allocation of the BRG matrix was given to us, but the matrix for the resultant RGBA wasn't. To complete it, we've added another cudaMalloc using uchar of 4 bytes.



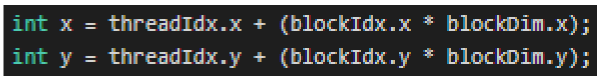
```
CU_CHECK2(cudaMalloc(d_brg, sizeof(uchar3)*width*height), "Alloc d_brg:");
// Can you finish this one? Replace cudaSucces with the proper cuda API call
CU_CHECK2(cudaMalloc(d_rgba, sizeof(uchar4)*width*height), "Alloc d_rgba:");
```

An important part of allocating memory, is freeing it later, so we've added the call to the cudaFree function for our new allocation.



```
CU_CHECK2(cudaFree(d_brg), "Cuda free d_bgr:");
CU_CHECK2(cudaFree(d_rgba), "Cuda free d_rgba:");
```

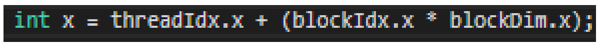
Last but not least, the coordinates used to access the BRG and RGBA arrays where implemented. For this we have used the indexes that came with the threads.



```
int x = threadIdx.x + (blockIdx.x * blockDim.x);
int y = threadIdx.y + (blockIdx.y * blockDim.y);
```

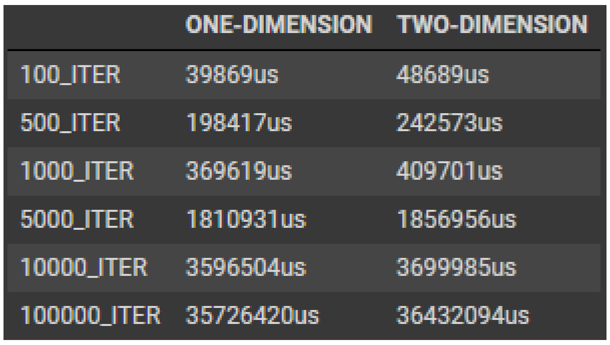
Section 2: Implement a single dimension Cuda GRID.

Using a single dimension GRID means that we only need one single coordinate to access our matrix. The modification is as simple as removing the 'y' coordinate, leaving the following code:



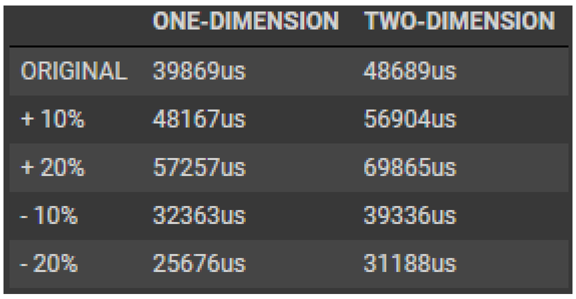
```
int x = threadIdx.x + (blockIdx.x * blockDim.x);
```

Running some experiments, changing the number of iterations while keeping the same block size (256), has resulted in the one-dimension kernel being quite faster than the two-dimension kernel. The results have been the following.



	ONE-DIMENSION	TWO-DIMENSION
100_ITER	39869us	48689us
500_ITER	198417us	242573us
1000_ITER	369619us	409701us
5000_ITER	1810931us	1856956us
10000_ITER	3596504us	3699985us
100000_ITER	35726420us	36432094us

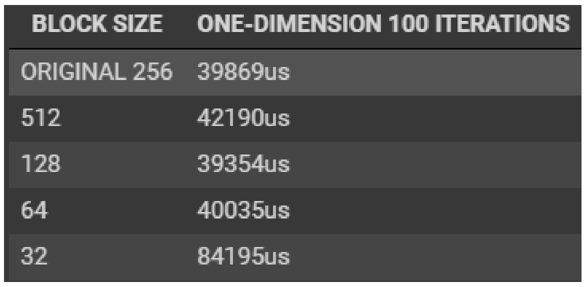
In addition to trying different iterations, we've also experimented changing the image size, from the original size, we've tested from -20% the original value up to 20%. The results have been the following.



	ONE-DIMENSION	TWO-DIMENSION
ORIGINAL	39869us	48689us
+ 10%	48167us	56904us
+ 20%	57257us	69865us
- 10%	32363us	39336us
- 20%	25676us	31188us

The single dimension kernel has been again the more efficient one in all the cases.

Finally, we've also experimented with the block size. The results have been the following:



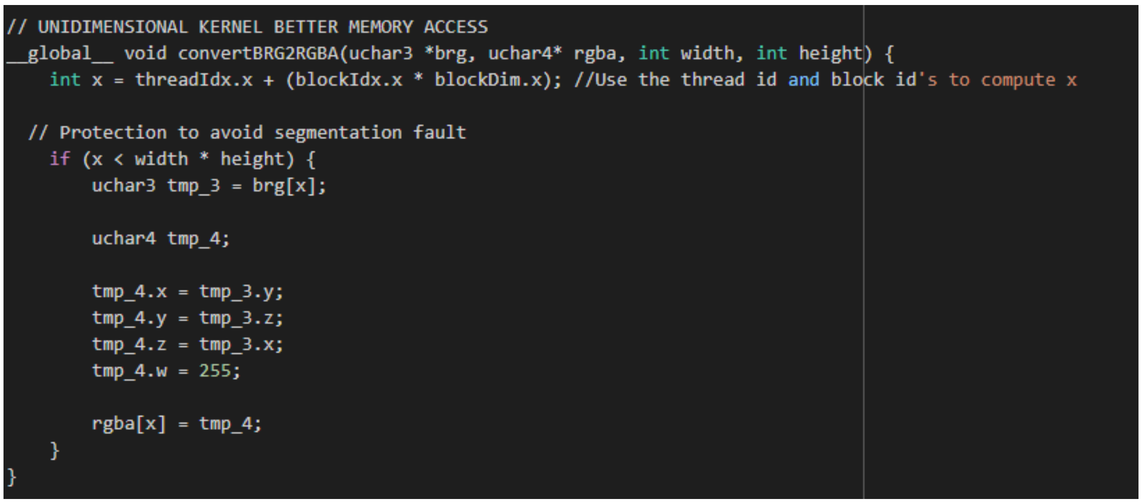
BLOCK SIZE	ONE-DIMENSION 100 ITERATIONS
ORIGINAL 256	39869us
512	42190us
128	39354us
64	40035us
32	84195us

The lowest execution time has been obtained using 128 as block size. Since the time reduction wasn't too big, we've decided to keep the original block size for the rest of the experiments in this lab.

Section 3: Optimizing memory accesses without using shared memory.

In this section we've applied the same technique as in section 3 of the OpenMP lab. We can access up to 32 bytes of at least 4 bytes each. Since our BRG matrix has 3 bytes, our memory access is not being efficient.

To try and make it more efficient, our solution first converts the BRG uchar3 into a uchar4 data type (converting the BRG values to RGBA and adding 255 to the alpha channel) and then assigning the RGBA matrix the uchar4 value directly, without the need of having to access each component of the uchar3 sequentially and thus having a more efficient access to memory.

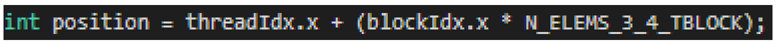


```
// UNIDIMENSIONAL KERNEL BETTER MEMORY ACCESS
__global__ void convertBRG2RGBA(uchar3 *brg, uchar4* rgba, int width, int height) {
    int x = threadIdx.x + (blockIdx.x * blockDim.x); //Use the thread id and block id's to compute x

  // Protection to avoid segmentation fault
    if (x < width * height) {
        uchar3 tmp_3 = brg[x];

        uchar4 tmp_4;

        tmp_4.x = tmp_3.y;
        tmp_4.y = tmp_3.z;
        tmp_4.z = tmp_3.x;
        tmp_4.w = 255;

        rgba[x] = tmp_4;
    }
}
```

Section 4: Optimizing memory accesses using shared memory.

Since the memory is going to be shared in this section, the access we implemented in Section 2 is no longer usable. We aren't accessing by block size anymore, instead we are trying to get 3 bytes of data out of a 4-byte memory access.

A constant that computed the 4-byte-elements of block_size-elements of 3 bytes was already provided, so the only we had to do was use it in our position access. Like this:



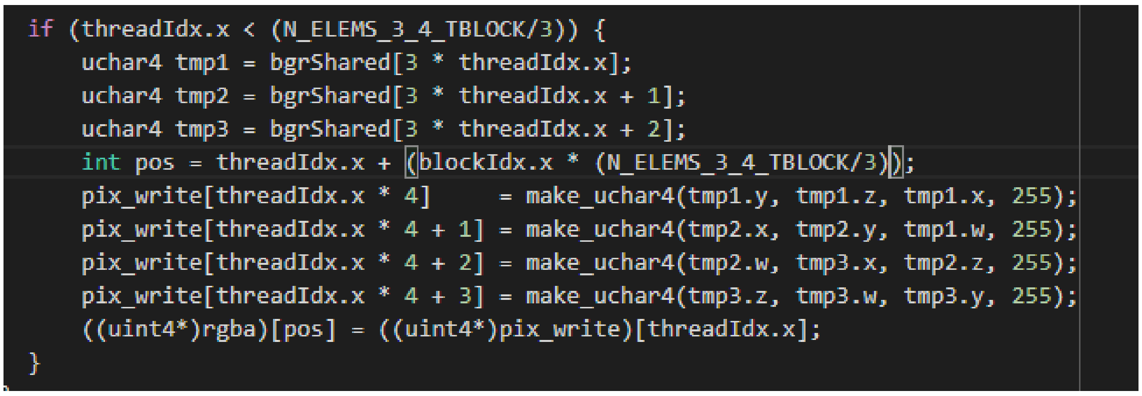
```
int position = threadIdx.x + (blockIdx.x * N_ELEMS_3_4_TBLOCK);
```

Section 5: Memory access algorithm.

As we've understood with the example in the instruction's pdf of this lab, 3 threads accessing directly the shared memory will collide. Each uchar3 occupies 3 of the 4 bytes that are accessed, this means that only the first thread (T0) will get all its data in one go, while the other threads (T1 and T2) will need 2 accesses to recover all their belonging data.

To avoid this, we can access the data as 12 bytes bank memory. This will result in less conflicts but will imply some control to avoid segmentation issues (because now with each "memory access" we'll be performing the equivalent to the memory accesses that 3 threads would do without the algorithm).

Coming from section 4, we need to divide the number of 4 bytes elements (N_ELEMS_3_4_TBLOCK) by the number of threads whom memory accesses our algorithm will be replacing; in our case 3. Having a block size of 256, leaves us with 192 4 byte elements, which in the end means that only 64 threads will perform the memory accesses. Like this:

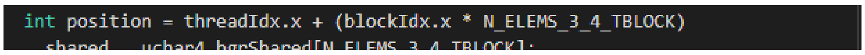


```
if (threadIdx.x < (N_ELEMS_3_4_TBLOCK/3)) {
    uchar4 tmp1 = bgrShared[3 * threadIdx.x];
    uchar4 tmp2 = bgrShared[3 * threadIdx.x + 1];
    uchar4 tmp3 = bgrShared[3 * threadIdx.x + 2];
    int pos = threadIdx.x + (blockIdx.x * (N_ELEMS_3_4_TBLOCK/3));
    pix_write[threadIdx.x * 4]     = make_uchar4(tmp1.y, tmp1.z, tmp1.x, 255);
    pix_write[threadIdx.x * 4 + 1] = make_uchar4(tmp2.x, tmp2.y, tmp1.w, 255);
    pix_write[threadIdx.x * 4 + 2] = make_uchar4(tmp2.w, tmp3.x, tmp2.z, 255);
    pix_write[threadIdx.x * 4 + 3] = make_uchar4(tmp3.z, tmp3.w, tmp3.y, 255);
    ((uint4*)rgba)[pos] = ((uint4*)pix_write)[threadIdx.x];
}
```

Section 6: Change the code to use CUDA streams.

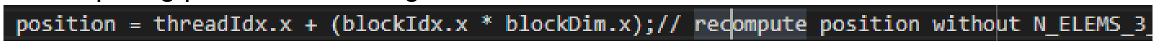
The modifications we've to the code have been the following:

- Computing position using N_ELEMS_3_4_TBLOCK:



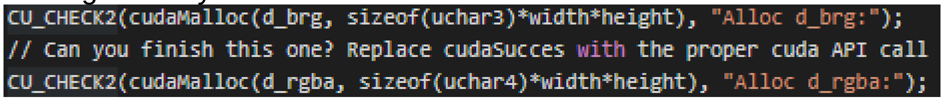
```
int position = threadIdx.x + (blockIdx.x * N_ELEMS_3_4_TBLOCK)
  shared   uchar4 bgrShared[N_ELEMS_3_4_TBLOCK];
```

- Recomputing position for writing the results:



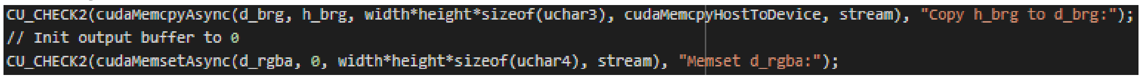
```
position = threadIdx.x + (blockIdx.x * blockDim.x);// recompute position without N_ELEMS_3_
```

- Allocating memory for the RGBA matrix:



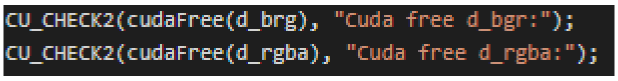
```
CU_CHECK2(cudaMalloc(d_brg, sizeof(uchar3)*width*height), "Alloc d_brg:");
// Can you finish this one? Replace cudaSucces with the proper cuda API call
CU_CHECK2(cudaMalloc(d_rgba, sizeof(uchar4)*width*height), "Alloc d_rgba:");
```

- Initializing GPU variables:



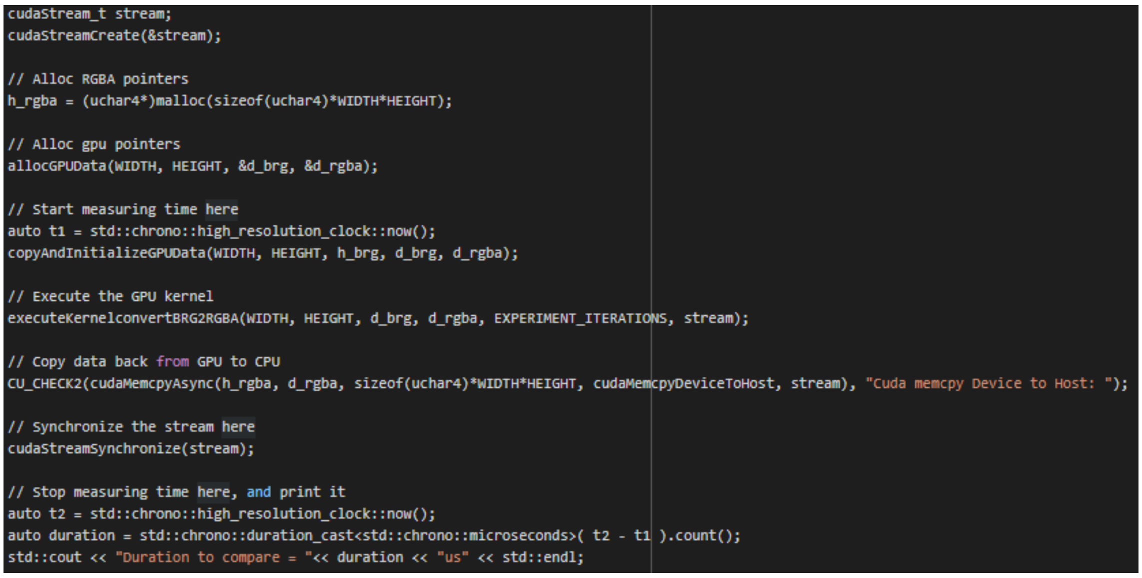
```
CU_CHECK2(cudaMemcpyAsync(d_brg, h_brg, width*height*sizeof(uchar3), cudaMemcpyHostToDevice, stream), "Copy h_brg to d_brg:");
// Init output buffer to 0
CU_CHECK2(cudaMemsetAsync(d_rgba, 0, width*height*sizeof(uchar4), stream), "Memset d_rgba:");
```

- Freeing allocated memory:



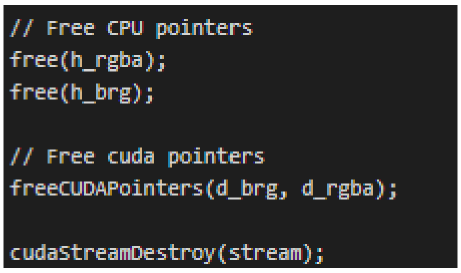
```
CU_CHECK2(cudaFree(d_brg), "Cuda free d_bgr:");
CU_CHECK2(cudaFree(d_rgba), "Cuda free d_rgba:");
```

- Add chrono code for stop measuring code, added stream to the function ‘*executeKernelconvertBRG2RGBA*’, and the stream’s synchronize function.



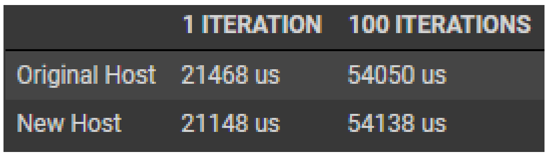
```
cudaStream_t stream;
cudaStreamCreate(&stream);

// Alloc RGBA pointers
h_rgba = (uchar4*)malloc(sizeof(uchar4)*WIDTH*HEIGHT);

// Alloc gpu pointers
allocGPUData(WIDTH, HEIGHT, &d_brg, &d_rgba);

// Start measuring time here
auto t1 = std::chrono::high_resolution_clock::now();
copyAndInitializeGPUData(WIDTH, HEIGHT, h_brg, d_brg, d_rgba);

// Execute the GPU kernel
executeKernelconvertBRG2RGBA(WIDTH, HEIGHT, d_brg, d_rgba, EXPERIMENT_ITERATIONS, stream);

// Copy data back from GPU to CPU
CU_CHECK2(cudaMemcpyAsync(h_rgba, d_rgba, sizeof(uchar4)*WIDTH*HEIGHT, cudaMemcpyDeviceToHost, stream), "Cuda memcpy Device to Host: ");

// Synchronize the stream here
cudaStreamSynchronize(stream);

// Stop measuring time here, and print it
auto t2 = std::chrono::high_resolution_clock::now();
auto duration = std::chrono::duration_cast<std::chrono::microseconds>( t2 - t1 ).count();
std::cout << "Duration to compare = "<< duration << "us" << std::endl;
```

- Stream destroy function:



```
// Free CPU pointers
free(h_rgba);
free(h_brg);

// Free cuda pointers
freeCUDAPointers(d_brg, d_rgba);

cudaStreamDestroy(stream);
```

Comparing the original host code with the new one, the execution times have been the following:



	1 ITERATION	100 ITERATIONS
Original Host	21468 us	54050 us
New Host	21148 us	54138 us

The new host code is quicker than the original one. We think this reduction in the execution time comes thanks to the usage of the Cuda streams. By using them, the GPU organized the tasks in the queue on the go as the resources become free, thus being more efficient and achieving shorter execution times.