Exercise (*Will not collect*)

Start with the demo code: `~zxu2/Public/CUDA_demo/test_1d_grid_2d_block.cu`, and revise the code to do following:

- Use 2D grid and 2D block to index elements in 2D matrix.
- The dynamic memory allocated for the 2D matrix is linear so that elements of the matrix is stored row by row. You can also use `cudaMallocPitch()` for memory allocation. See `~zxu2/Public/CUDA_demo/pitch_sample.cu` for example.
- Implement a kernel to do matrix addition.
- Assume that the dimension of the matrix is 6 by 12. Assume that the grid dimension is 2 by 3 and block dimension is 3 by 4.
- Consider how to handle the case in which dimensions of the matrices are smaller/greater than dimensions of the thread topology.

The project is on the next page.

CUDA project 01

The goal of this project is to practice CUDA programming on GPUs. Specifically, we implement kernel functions for image processing, and experiment on using different thread grid configuration for running these kernel functions.

The `~zxu2/Public/CUDA_demo/GPU_proj01` can be used as the base code for this project.

Compiling options:

1. Set up the CUDA environment. See Pg. 13 of the first GPU notes.

2) `nvcc matr_multiply.cu rw_jpg.cpp  main.cpp Vector.cpp -ljpeg -std=c++11 –o prog`

3) or use the `Makefile (type at command line: make)`

`main()` function in `main.cpp`

To run the program: `./prog Sunflower.jpg`

What's done on CPU:

1. The current code read in this image by calling the function

```
read_jpg(filename, &RGB_bundle);
```

On return, the `RGB_bundle struct` stores the RGB values of the image by array pointed by member pointer `image_data` of the `struct`.

The double for loop shows how to extract RGB values for individual pixels.

2. Use matrices (`Red, Blue, Green`) to store RGB values respectively. Each matrix only saves one color.

3. Use the following 3 by 3 filter to blur the image:

1/9{{1 1 1}; {1 1 1}; {1 1 1}}.

Multiply each element of the kernel with its corresponding element of the image matrix (the one which is overlapped with it)

Sum up all product outputs and put the result in the output matrix.

4. Write the blurred image to a file by call the function `write_jpg(const char*,bundle*);`

5. Function `cuda_kernel_wrapper()` is declared in `cuda_wrapper.h`, and defined in `matr_multiply.cu`. It is a C function interfacing the C++ code and the CUDA code. Kernels are eventually launched by subsequent calls inside this function. It is called by `main()`.

6. The `Vector` class is not necessary for this project.

7. `matr_multiply.cu` was originally designed for doing matrix-matrix multiplication on GPU.

Instructions of computer implementation:

1. Modify arguments of `cuda_kernel_wrapper()` to pass matrices (Red, Blue, Green).

2. If you wish to use `struct cu_Matrix` to store rgb values of the image, change

```
typedef struct {
    int width;
    int height;
    float* elements;
} cu_Matrix;
```

to

```
typedef struct {
    int             width;
    int             height;
    unsigned char* elements;
} cu_Matrix;
```

3. Use 2D block decomposition on the image pixels to yield parallel tasks.

4. According to 2D block decomposition of tasks, use 2D grid of blocks and 2D block of threads structure to launch kernel function.

5. The kernel function does the filtering. Each thread filters one pixel of the image.

6. Banchmark the GPU time used by the kernel.

7. (Bonus. This part can be submitted after discussing shared memory). Can shared memory be used to improve the performance for the kernel function? Implement the kernel function by using the shared memory. Since the size of the shared memory is limited (48KB), if the matrix size is arbitrarily large, how should shared memory be used? Can you use padded memory by `cudaMallocPitch()` instead?

Test the code.

1. Let the number of threads within a thread block be a multiple of 32. For example, 64 or 256.
2. You can also use jpeg format images of different sizes to test the code. See if GPU times vary as image sizes.
3. To measure the performance of the GPU kernel execution, use the following code:

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);

/// your kernel call here

cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);

float elapseTime;
cudaEventElapsedTime(&elapseTime, start, stop);
printf("Time to run the kernel: %f ms\n", elapseTime);
```

4. We can also use `nvprof` to profile the code.
5. Submit your code together with a report containing the code performance results.