Subject-Part2-en.md

Assignment Part Two: blur and convolutions

In this homework we are blurring an image. To do this, imagine that we have a square array of weight values. For each pixel in the image, imagine that we overlay this square array of weights on top of the image such that the center of the weight array is aligned with the current pixel. To compute a blurred pixel value, we multiply each pair of numbers that line up. In other words, we multiply each weight with the pixel underneath it. Finally, we add up all of the multiplied numbers and assign that value to our output for the current pixel. We repeat this process for all the pixels in the image.

To help get you started, we have included some useful notes here.

Structure transformation

For a color image that has multiple channels, we suggest separating the different color channels so that each color is stored contiguously instead of being interleaved. This will simplify your code.

That is instead of RGBARGBARGBARGBA... we suggest transforming to three arrays (as in the previous homework we ignore the alpha channel again):

1. RRRRRRRR...
2. GGGGGGGG...
3. BBBBBBBB...

The original layout is known an Array of Structures (AoS) whereas the format we are converting to is known as a Structure of Arrays (SoA).

As a warm-up, we will ask you to write the kernel that performs this separation. You should then write the "meat" of the assignment, which is the kernel that performs the actual blur. We provide code that re-combines your blurred results for each color channel.

Convolutions

You must fill in the gaussian_blur kernel to perform the blurring of the inputChannel, using the array of weights, and put the result in the outputChannel.

Here is an example of computing a blur, using a weighted average, for a single pixel in a small image.

Array of weights:

0.0	0.2	0.0
0.2	0.2	0.2
0.0	0.2	0.0

Image (note that we align the array of weights to the center of the box):

1	2	5	2	0	3			
	---	---	---					
3	|2	5	1|	6	0		0.0*2 + 0.2*5 + 0.0*1 +	
4	|3	6	2|	1	4	->	0.2*3 + 0.2*6 + 0.2*2 +	-> 3.2
0	|4	0	3|	4	2		0.0*4 + 0.2*0 + 0.0*3	
	---	---	---					

9	6	5	0	3	9			
		(1)					(2)	(3)

Getting started

A good starting place is to map each thread to a pixel as you have before. Then every thread can perform steps 2 and 3 in the diagram above completely independently of one another.

Note that the array of weights is square, so its height is the same as its width. We refer to the array of weights as a filter, and we refer to its width with the variable filterWidth.

Also note that we've supplied a helpful debugging function called checkCudaErrors. You should wrap your allocation and copying statements like we've done in the code we're supplying you. Here is an example of the unsafe way to allocate memory on the GPU:

```
cudaMalloc(&d\_red, sizeof(unsigned char) * numRows * numCols);
```

Here is an example of the safe way to do the same thing:

```
checkCudaErrors(cudaMalloc(&d\_red, sizeof(unsigned char) * numRows * numCols));
```

Writing code the safe way requires slightly more typing, but is very helpful for catching mistakes. If you write code the unsafe way and you make a mistake, then any subsequent kernels won't compute anything, and it will be hard to figure out why. Writing code the safe way will inform you as soon as you make a mistake.

Finally, remember to free the memory you allocate at the end of the function.