1. *// Homework 2*
2. *// Image Blurring*
3. *//*
4. *// In this homework we are blurring an image. To do this, imagine that we have*
5. *// a square array of weight values. For each pixel in the image, imagine that we*
6. *// overlay this square array of weights on top of the image such that the center*
7. *// of the weight array is aligned with the current pixel. To compute a blurred*
8. *// pixel value, we multiply each pair of numbers that line up. In other words, we*
9. *// multiply each weight with the pixel underneath it. Finally, we add up all of the*
10. *// multiplied numbers and assign that value to our output for the current pixel.*
11. *// We repeat this process for all the pixels in the image.*
13. *// To help get you started, we have included some useful notes here.*
15. *//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**
17. *// For a color image that has multiple channels, we suggest separating*
18. *// the different color channels so that each color is stored contiguously*
19. *// instead of being interleaved. This will simplify your code.*
21. *// That is instead of RGBARGBARGBARGBA... we suggest transforming to three*
22. *// arrays (as in the previous homework we ignore the alpha channel again):*
23. *//  1) RRRRRRRR...*
24. *//  2) GGGGGGGG...*
25. *//  3) BBBBBBBB...*
26. *//*
27. *// The original layout is known an Array of Structures (AoS) whereas the*
28. *// format we are converting to is known as a Structure of Arrays (SoA).*
30. *// As a warm-up, we will ask you to write the kernel that performs this*
31. *// separation. You should then write the "meat" of the assignment,*
32. *// which is the kernel that performs the actual blur. We provide code that*
33. *// re-combines your blurred results for each color channel.*
35. *//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**
37. *// You must fill in the gaussian\_blur kernel to perform the blurring of the*
38. *// inputChannel, using the array of weights, and put the result in the outputChannel.*
40. *// Here is an example of computing a blur, using a weighted average, for a single*
41. *// pixel in a small image.*
42. *//*
43. *// Array of weights:*
44. *//*
45. *//  0.0  0.2  0.0*
46. *//  0.2  0.2  0.2*
47. *//  0.0  0.2  0.0*
48. *//*
49. *// Image (note that we align the array of weights to the center of the box):*
50. *//*
51. *//    1  2  5  2  0  3*
52. *//       -------*
53. *//    3 |2  5  1| 6  0       0.0\*2 + 0.2\*5 + 0.0\*1 +*
54. *//      |       |*
55. *//    4 |3  6  2| 1  4   ->  0.2\*3 + 0.2\*6 + 0.2\*2 +   ->  3.2*
56. *//      |       |*
57. *//    0 |4  0  3| 4  2       0.0\*4 + 0.2\*0 + 0.0\*3*
58. *//       -------*
59. *//    9  6  5  0  3  9*
60. *//*
61. *//         (1)                         (2)                 (3)*
62. *//*
63. *// A good starting place is to map each thread to a pixel as you have before.*
64. *// Then every thread can perform steps 2 and 3 in the diagram above*
65. *// completely independently of one another.*
67. *// Note that the array of weights is square, so its height is the same as its width.*
68. *// We refer to the array of weights as a filter, and we refer to its width with the*
69. *// variable filterWidth.*
71. *//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**
73. *// Your homework submission will be evaluated based on correctness and speed.*
74. *// We test each pixel against a reference solution. If any pixel differs by*
75. *// more than some small threshold value, the system will tell you that your*
76. *// solution is incorrect, and it will let you try again.*
78. *// Once you have gotten that working correctly, then you can think about using*
79. *// shared memory and having the threads cooperate to achieve better performance.*
81. *//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**
83. *// Also note that we've supplied a helpful debugging function called checkCudaErrors.*
84. *// You should wrap your allocation and copying statements like we've done in the*
85. *// code we're supplying you. Here is an example of the unsafe way to allocate*
86. *// memory on the GPU:*
87. *//*
88. *// cudaMalloc(&d\_red, sizeof(unsigned char) \* numRows \* numCols);*
89. *//*
90. *// Here is an example of the safe way to do the same thing:*
91. *//*
92. *// checkCudaErrors(cudaMalloc(&d\_red, sizeof(unsigned char) \* numRows \* numCols));*
93. *//*
94. *// Writing code the safe way requires slightly more typing, but is very helpful for*
95. *// catching mistakes. If you write code the unsafe way and you make a mistake, then*
96. *// any subsequent kernels won't compute anything, and it will be hard to figure out*
97. *// why. Writing code the safe way will inform you as soon as you make a mistake.*
99. *// Finally, remember to free the memory you allocate at the end of the function.*
101. *//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**
103. #include "reference\_calc.cpp"
104. #include "utils.h"
106. \_\_global\_\_
107. void gaussian\_blur(const unsigned char\* const inputChannel,
108. unsigned char\* const outputChannel,
109. int numRows, int numCols,
110. const float\* const filter, const int filterWidth)
111. {
112. *// TODO*
114. *// NOTE: Be sure to compute any intermediate results in floating point*
115. *// before storing the final result as unsigned char.*
117. *// NOTE: Be careful not to try to access memory that is outside the bounds of*
118. *// the image. You'll want code that performs the following check before accessing*
119. *// GPU memory:*
120. *//*
121. *// if ( absolute\_image\_position\_x >= numCols ||*
122. *//      absolute\_image\_position\_y >= numRows )*
123. *// {*
124. *//     return;*
125. *// }*
127. *// NOTE: If a thread's absolute position 2D position is within the image, but some of*
128. *// its neighbors are outside the image, then you will need to be extra careful. Instead*
129. *// of trying to read such a neighbor value from GPU memory (which won't work because*
130. *// the value is out of bounds), you should explicitly clamp the neighbor values you read*
131. *// to be within the bounds of the image. If this is not clear to you, then please refer*
132. *// to sequential reference solution for the exact clamping semantics you should follow.*
133. assert(filterWidth % 2 == 1);
135. const int2 thread\_2D\_pos = make\_int2( blockIdx.x \* blockDim.x + threadIdx.x,
136. blockIdx.y \* blockDim.y + threadIdx.y);
137. const int thread\_1D\_pos = thread\_2D\_pos.y \* numCols + thread\_2D\_pos.x;
139. if (thread\_2D\_pos.x >= numCols || thread\_2D\_pos.y >= numRows)
140. return;
142. int c = thread\_2D\_pos.x;
143. int r = thread\_2D\_pos.y;
145. float result = 0.f;
146. *//For every value in the filter around the pixel (c, r)*
147. for (int filter\_r = -filterWidth/2; filter\_r <= filterWidth/2; ++filter\_r) {
148. for (int filter\_c = -filterWidth/2; filter\_c <= filterWidth/2; ++filter\_c) {
149. *//Find the global image position for this filter position*
150. *//clamp to boundary of the image*
151. int image\_r = min(max(r + filter\_r, 0), static\_cast<int>(numRows - 1));
152. int image\_c = min(max(c + filter\_c, 0), static\_cast<int>(numCols - 1));
154. float image\_value = static\_cast<float>(inputChannel[image\_r \* numCols + image\_c]);
155. float filter\_value = filter[(filter\_r + filterWidth/2) \* filterWidth + filter\_c + filterWidth/2];
157. result += image\_value \* filter\_value;
158. }
159. }
160. outputChannel[thread\_1D\_pos] = result;
161. }
163. *//This kernel takes in an image represented as a uchar4 and splits*
164. *//it into three images consisting of only one color channel each*
165. \_\_global\_\_
166. void separateChannels(const uchar4\* const inputImageRGBA,
167. int numRows,
168. int numCols,
169. unsigned char\* const redChannel,
170. unsigned char\* const greenChannel,
171. unsigned char\* const blueChannel)
172. {
173. *// TODO*
174. *//*
175. *// NOTE: Be careful not to try to access memory that is outside the bounds of*
176. *// the image. You'll want code that performs the following check before accessing*
177. *// GPU memory:*
178. *//*
179. *// if ( absolute\_image\_position\_x >= numCols ||*
180. *//      absolute\_image\_position\_y >= numRows )*
181. *// {*
182. *//     return;*
183. *// }*
184. const int2 thread\_2D\_pos = make\_int2( blockIdx.x \* blockDim.x + threadIdx.x,
185. blockIdx.y \* blockDim.y + threadIdx.y);
187. const int thread\_1D\_pos = thread\_2D\_pos.y \* numCols + thread\_2D\_pos.x;
189. if (thread\_2D\_pos.x >= numCols || thread\_2D\_pos.y >= numRows)
190. return;
192. uchar4 inputPixel = inputImageRGBA[thread\_1D\_pos];
193. unsigned char red   = inputPixel.x;
194. unsigned char green = inputPixel.y;
195. unsigned char blue  = inputPixel.z;
197. redChannel[thread\_1D\_pos] = red;
198. greenChannel[thread\_1D\_pos] = green;
199. blueChannel[thread\_1D\_pos] = blue;
200. }
202. *//This kernel takes in three color channels and recombines them*
203. *//into one image.  The alpha channel is set to 255 to represent*
204. *//that this image has no transparency.*
205. \_\_global\_\_
206. void recombineChannels(const unsigned char\* const redChannel,
207. const unsigned char\* const greenChannel,
208. const unsigned char\* const blueChannel,
209. uchar4\* const outputImageRGBA,
210. int numRows,
211. int numCols)
212. {
213. const int2 thread\_2D\_pos = make\_int2( blockIdx.x \* blockDim.x + threadIdx.x,
214. blockIdx.y \* blockDim.y + threadIdx.y);
216. const int thread\_1D\_pos = thread\_2D\_pos.y \* numCols + thread\_2D\_pos.x;
218. *//make sure we don't try and access memory outside the image*
219. *//by having any threads mapped there return early*
220. if (thread\_2D\_pos.x >= numCols || thread\_2D\_pos.y >= numRows)
221. return;
223. unsigned char red   = redChannel[thread\_1D\_pos];
224. unsigned char green = greenChannel[thread\_1D\_pos];
225. unsigned char blue  = blueChannel[thread\_1D\_pos];
227. *//Alpha should be 255 for no transparency*
228. uchar4 outputPixel = make\_uchar4(red, green, blue, 255);
230. outputImageRGBA[thread\_1D\_pos] = outputPixel;
231. }
233. unsigned char \*d\_red, \*d\_green, \*d\_blue;
234. float         \*d\_filter;
236. void allocateMemoryAndCopyToGPU(const size\_t numRowsImage, const size\_t numColsImage,
237. const float\* const h\_filter, const size\_t filterWidth)
238. {
240. *//allocate memory for the three different channels*
241. *//original*
242. checkCudaErrors(cudaMalloc(&d\_red,   sizeof(unsigned char) \* numRowsImage \* numColsImage));
243. checkCudaErrors(cudaMalloc(&d\_green, sizeof(unsigned char) \* numRowsImage \* numColsImage));
244. checkCudaErrors(cudaMalloc(&d\_blue,  sizeof(unsigned char) \* numRowsImage \* numColsImage));
246. *//TODO:*
247. *//Allocate memory for the filter on the GPU*
248. *//Use the pointer d\_filter that we have already declared for you*
249. *//You need to allocate memory for the filter with cudaMalloc*
250. *//be sure to use checkCudaErrors like the above examples to*
251. *//be able to tell if anything goes wrong*
252. *//IMPORTANT: Notice that we pass a pointer to a pointer to cudaMalloc*
253. checkCudaErrors(cudaMalloc(&d\_filter,  sizeof(float) \* filterWidth \* filterWidth));
254. *//TODO:*
255. *//Copy the filter on the host (h\_filter) to the memory you just allocated*
256. *//on the GPU.  cudaMemcpy(dst, src, numBytes, cudaMemcpyHostToDevice);*
257. *//Remember to use checkCudaErrors!*
258. checkCudaErrors(cudaMemcpy(d\_filter, h\_filter, sizeof(float) \* filterWidth \* filterWidth, cudaMemcpyHostToDevice));
259. }
261. void your\_gaussian\_blur(const uchar4 \* const h\_inputImageRGBA, uchar4 \* const d\_inputImageRGBA,
262. uchar4\* const d\_outputImageRGBA, const size\_t numRows, const size\_t numCols,
263. unsigned char \*d\_redBlurred,
264. unsigned char \*d\_greenBlurred,
265. unsigned char \*d\_blueBlurred,
266. const int filterWidth)
267. {
268. *//TODO: Set reasonable block size (i.e., number of threads per block)*
269. const dim3 blockSize(16, 16, 1);
271. *//TODO:*
272. *//Compute correct grid size (i.e., number of blocks per kernel launch)*
273. *//from the image size and and block size.*
274. const dim3 gridSize((numCols/16) + 1, (numRows/16) + 1, 1);
276. *//TODO: Launch a kernel for separating the RGBA image into different color channels*
277. separateChannels<<<gridSize, blockSize>>>(d\_inputImageRGBA, numRows, numCols, d\_red, d\_green, d\_blue);
279. *// Call cudaDeviceSynchronize(), then call checkCudaErrors() immediately after*
280. *// launching your kernel to make sure that you didn't make any mistakes.*
281. cudaDeviceSynchronize(); checkCudaErrors(cudaGetLastError());
283. *//TODO: Call your convolution kernel here 3 times, once for each color channel.*
284. gaussian\_blur<<<gridSize, blockSize>>>(d\_red, d\_redBlurred, numRows, numCols, d\_filter, filterWidth);
285. gaussian\_blur<<<gridSize, blockSize>>>(d\_green, d\_greenBlurred, numRows, numCols, d\_filter, filterWidth);
286. gaussian\_blur<<<gridSize, blockSize>>>(d\_blue, d\_blueBlurred, numRows, numCols, d\_filter, filterWidth);
288. *// Again, call cudaDeviceSynchronize(), then call checkCudaErrors() immediately after*
289. *// launching your kernel to make sure that you didn't make any mistakes.*
290. cudaDeviceSynchronize(); checkCudaErrors(cudaGetLastError());
292. *// Now we recombine your results. We take care of launching this kernel for you.*
293. *//*
294. *// NOTE: This kernel launch depends on the gridSize and blockSize variables,*
295. *// which you must set yourself.*
296. recombineChannels<<<gridSize, blockSize>>>(d\_redBlurred,
297. d\_greenBlurred,
298. d\_blueBlurred,
299. d\_outputImageRGBA,
300. numRows,
301. numCols);
302. cudaDeviceSynchronize(); checkCudaErrors(cudaGetLastError());
303. }

306. *//Free all the memory that we allocated*
307. *//TODO: make sure you free any arrays that you allocated*
308. void cleanup() {
309. checkCudaErrors(cudaFree(d\_red));
310. checkCudaErrors(cudaFree(d\_green));
311. checkCudaErrors(cudaFree(d\_blue));
312. }