Report for CSC367H5 Parallel Programming

Assignment 4

Name: Lang Qin Utorid: 1005348982

For all kernel invocations to process PGM format images, each of them will have three main stages which corresponding three separate kernels. First is called *kernel(n)* which is the kernel to process image pixel(s). The second is called *reduction.* This kernel will process the updated images from last kernel to find minimum and maximum pixel values. There will be two global arrays to store the min and max. For single *reduction* call, it will create *m* blocks such that each thread will process one element in the arrays. Since we are using reduction strategy, all of threads in one block will calculate the min and max for this block. It ends up with updating global arrays while reduce the size of the arrays we need to process into *m*. Thus, this kernel will be put in a while loop to keep updating global min and max arrays until the number of blocks becomes 1. That is, the first element of both of them will be the final minimum and maximum pixel values. The final kernel is called *normalize(n)* which is the kernel to normalize the output image given the global minimum and maximum pixel values obtained from *reduction* kernel. Since there will be implicit barriers between each kernel calls, synchronization between each stage is ensured.

Each kernel invocation will have different memory access pattern and data processing strategies. That is, how they access and process the image pixels. For each kernel innovation will have different implementations of *kernel(n)* and *normalize(n).* We will inspect different features and performance for each of them. Note that we

define the maximum threads for one block is 512 and maximum pixels each thread will process are 16. We choose the image size to test performance on to be 10M. The result comes in figure 1.

CPU_time(ms)	Kernel	GPU_time(ms)	TransferIn(ms)	TransferOut(ms)	Speedup_noTrf	Speedup
142.994080	1	35.015938	15.744224	15.114720	4.08	2.17
142.994080	2	5.313376	31.661568	21.539392	26.91	2.44
142.994080	3	23.574112	9.005760	13.944160	6.07	3.07
142.994080	4	5.272640	14.053088	24.234880	27.12	3.28
142.994080	5	3.437216	8.749536	13.752288	41.60	5.51

Figure 1

1) kernel1, normalize1:

   At this kernel, we assign each thread one single pixel to process and access the input image matrix in column major way. From figure 1 we can see that compared with CPU execution time which is 142.99 ms, it has 35.01 ms of GPU execution time, which has a speed up rate of 4.08 if not including transfer time. We can see the GPU shows excellent advantage in doing jobs like image processing. However, among all kernels, it is the slowest one. Not only because it has each thread only process one pixel, but also it accesses the image matrix in column major way. It should have bad data locality.

2) kernel2, normalize2:

   We still assign each thread one pixel to work on, but it will access the matrix in row major. Compared with kernel1, it should increase performance since it has good data locality. And we can see that the running time reduced to only about 5.31 ms.

3) kernel3, normalize3:

   This time we assign each thread to process 16 pixels, consecutive rows and row

major. Thus, each thread will have a chunk size of 16 to work on and all of elements in one chunk are consecutive in one row. The running time is 23.57 ms. Compared with kernel1, it has faster performance. However, it is much slower than kernel2.

4) Kernel4, normalize4:

   We still assign each thread 16 pixels to process, but this time each thread will sequentially accesses 16 pixels with a stride equal to the number of threads in one block. Performance for this kernel increases a lot compared with kernel3. It has a running time of 5.27 ms, which is also a bit faster than kernel2 and has a speed up rate of 3.28. Both of kernel3 and kernel4 works on 16 pixels per thread, the reason why kernel4 is much faster is because it satisfies memory coalescing. It accesses memory with a stride of number of threads. So, for each memory access, each thread will have consecutive access to the element of the matrix.

Among 4 kernels we have found that kernel4 is the fastest, so we will implement kernel5 base on it.

5) kernel5, reduction5, normalize5:

   Since we are using filter of dimension of 5 and all the elements in this filter are -1, except the one in the middle, which is 24 and at index 12. We change the processing strategy for single pixel such that if current index for the filter is not 12, we use -1. Otherwise, we use 24. This actually prevents the memory access to the global filter memory. Also, instead of modifying output matrix[i] at each

iteration step, we use an accumulator to accumulate the amount we want to apply to that pixel. After iteration is done, we assign matrix[i] equal to the accumulator. This will reduce the memory accesses to the global output matrix memory. Also, we add one reduction step at the end of the *kernel5* call. Now, besides from processing 16 pixels, each thread in *kernel5* will additionally produces local min and max for the pixels they have processed. And if we create *m* blocks to process pixels, we will end up with obtaining min and max arrays of size *m* after *kernel5* terminates. Compared with *kernel4* on the same setup, this strategy will reduce one reduction kernel call. Moreover, we have updated the reduction kernel into a more optimized version, called *reduction5*. In this version, we decrease divergence to let all threads do work at the start. And for all threads we enforce sequential access to reduce bank conflict. Also, we unroll all the for loops in this kernel call. The running time for this kernel implementation is only about 3.43 ms. Compared with *kernel4*, it improves performance by 34% when processing an image of size 10M. We want to see whether performance will change if processing an image of larger size. So, we construct a very big image, about 134 M. And run the tests for all kernels again on this image. The results are in figure 2.

CPU_time(ms)	Kernel	GPU_time(ms)	TransferIn(ms)	TransferOut(ms)	Speedup_noTrf	Speedup
2820.382324	1	389.173340	113.336759	181.166595	7.25	4.13
2820.382324	2	67.685280	126.504448	181.095520	41.67	7.52
2820.382324	3	257.575195	126.688225	183.629028	10.95	4.97
2820.382324	4	67.690620	127.961411	185.479355	41.67	7.40
2820.382324	5	37.865089	127.163971	184.490524	74.49	8.07

Figure 2

We can see that kernel5 still has the best performance, with a total speed up rate of 8.07.