Samuel Weismuller
WES 237B
9/4/2022

Assignment 4

Lab Part 1:

1. ex1.cu allocates and array of 4 elements and then transfers it into GPU memory. It then launches two blocks of two GPU threads. Each thread uses its block and thread indices to obtain a unique offset into the array and print the data at that location. The allocated GPU memory is then freed.
2. ex2.cu uses only one thread block. Instead of using the block index and thread index to determine the offset into the input array, the thread block is made two dimensional and the offset is found by the thread index's x and y value.
3. lw_managed.cu creates a shared memory block between the GPU and CPU meaning that data doesn't have to be explicitly copied between the two devices uses cudaMemcpy() like in lw.cu.

Lab Part 2:

Execution time (seconds) using an image size of 1024x1024:

	gray	invert	blur (10x10)
OpenCV	0.000719	0.000608	0.00438937
CPU	0.008826	0.001351	0.160571
GPU	0.006971	0.002343	0.0248763

OpenCV is the clear winner for the most optimal of all algorithms. The CPU even manages to beat out the GPU on inverting likely due to the invert being the least computationally taxing. However, it should be noted as the image sizes increase, the GPU algorithms scale fairly well. It's possible that, for a large enough image, the GPU algorithm might eventually overtake OpenCV.

Execution time (seconds) using blur on different image sizes:

	1024x1024	4096x4096	8192x8192
CPU	0.00438937	0.0112991	0.0354479
GPU	0.0248763	0.0379734	0.0607455

Assignment Part 1:

Execution time (ms) for the Sobel filter on square image sizes:

	512	1024	2048	4096
OpenCV	8.10878	29.4745	113.054	424.049
CPU	7.71855	29.7888	115.624	455.206
GPU	5.05632	10.2429	37.1279	70.3004

The GPU had the best performance of the three algorithms by far. The OpenCV and CPU algorithms were fairly comparable though the OpenCV algorithm scaled better.

Samuel Weismuller
WES 237B
9/4/2022

Assignment Part 2:

Speedup of block matrix multiplication for varying sizes of M and N and block size 16:

	M=16	M=32	M=128	M=512	M=1024
N=16	0.52	0.56	0.54	0.77	0.68
N=32	0.67	0.63	0.98	2.15	3.91
N=128	1.26	1.62	4.61	5.61	5.65
N=512	3.73	3.63	5.63	4.7	5.33
N=1024	4.08	4.38	4.81	5.16	11.47

Speed up improves as the matrix size increased. There was some randomness to the speedup values and running the matrix multiplication multiple times would result in the speedup increasing, likely due to caching somewhere or a process scheduling quirk. As a result, it is not clear if increasing M or N has a greater impact on the speedup as they appear roughly equivalent. Some instances of M or N being 128 were more performant that 512, particularly when N=128. This could be an artifact of the block size or just coincidence.

Speedup for various block sizes on 1024x1024 matrix:

	4	8	16	32
Speedup	4.7	9.46	11.47	13.28

It appears like the optimum block size for the 1024x1024 matrix is 32, which is as large of a square thread block as we can make.