WES 237B Lab/Assignment 4

Lab Part 1

Compile and run instructions

```
cd ./lab4/lab4-part1/

g++ main.cc -o hello_cpu
nvcc main.cu -o hello_gpu
./hello_cpu
./hello_gpu

nvcc ex1.cu -o ex1
nvcc ex2.cu -o ex2
./ex1
./ex2

nvcc lw.cu -o lw
nvcc lw_managed.cu -o lw_managed
./lw
./lw_managed
```

Deliverables

1.Explain in plain English what example 1 does.

- Example 1 sets up a basic CUDA kernel. This kernel runs on the GPU(device). The kernel accesses and prints values that are stored in an array `a` in the GPUs main memory. Each thread accesses one value of `a`, and the thread calculates which index to acces based on its blockId, ThreadId and the blockDim.

2.Explain the difference between example 1 and example 2. What is significant about it?

- Example 2 configures the GPU blocks and threads differently then Example 1. Example one used blocks and 2 threads per block to perform 4 parallel operations. Example 2 uses 1 block but 4 threads per block. Also example 2 stores its threads as a 2D (2x2) structure. As a result, Example 2 still has 4 total threads just like example 1, but the way the calculation for the index of a has changed. This is significant because it shows the importance of how the programmer decides to configure the device memory, Depending on the application how you layout the device memory can make for more or less efficient solutions.

2.Provide a copy of your myKernel() implementation from your matrix multiplication.

- Code is here: ./lab4/lab4-part1

3.Explain the difference between the two lw implementations.

- The first implementation uses manual memory management. Pointers for the device memory need so be declared and then CudaMalloc() is called to allocate GPU memory for the arrays. The programmer must manually copy data to and from the device using the cudaMemcpy() function and also free the device memory with CudaFree() when it is no longer needed. The second implementation uses managed memory, which automatically handles the memory management for you. You just call the cudaMallocManaged() function for each array you want on the device and then memory management is taken care of under the hood. This allows for less lines of code in your script since you can skip the cudaMemcpy() lines, but it also gives you slightly less control over the device and host memory. It also abstracts the fact that the Jetson actually has separate CPU and GPU memory.

Lab Part 2

Compile and run instructions

```
cd ./lab4/lab4-part2/
make

./lab4 arg
```

arg = 0 for openCV, 1 for CPU, 2 for GPU

Deliverables

1.Submit all of your final code for Part 2.

- All code is here: ./lab4/lab4-part2

2.For each algorithm (greyscale, inversion, blur), which implementation has the best performance? Make a quantitative case.

- OpenCV converts frames from rgb to grayscale in about 0.001 seconds on a 1024x1024 image. The CPU algorithm took about 0.03 seconds. The GPU algorithm without unified memory took about 0.006s. So the OpenCV function was the fastest at converting the rgb image to grayscale, followed by the GPU and then finally the CPU.
- OpenCV performs grayscale conversion + inversion in about 0.002 seconds. The CPU algorithm took about 0.03 seconds. The GPU algorithm without unified memory took about 0.008s. Subtracting out the time for just grayscale we get that openCV completed the inversion in 0.001 seconds, and the GPU finished in 0.002 seconds. Somehow the CPU time to convert to grayscale then invert was almost identical to the CPU time to just convert to grayscale. Again openCV is the fastest, but here the GPU is not too far behind.
- OpenCV performs grayscale conversion + blur in about 0.0035 seconds. The CPU algorithm took about 0.110 seconds. The GPU algorithm without unified memory took about 0.0132 seconds. Subtracting out the time for just grayscale we get that openCV completed the blur in 0.034 seconds,the CPU took around 0.080 seconds, and the GPU finished in 0.0072 seconds. Again openCV is the fastest.

Assignment Part 1: Sobel Filter

Compile and run instructions

```
cd ./sobel
make
./sobel arg1 arg2 arg3
```

arg1 and arg2 are width and height. arg3 is mode. 0 for openCV, 1 for CPU, 2 for GPU.

Deliverables

1. Report approximate execution times for OpenCV Sobel, CPU Sobel, and GPU Sobel for different image sizes.

Size	OpenCv	CPU	GPU
512x512	6.0-8.0 ms	7.1-7.6 ms	1.9-2.5 ms
1024x1024	26-29 ms	28 ms	3-4 ms
4096x4096	419 ms	453 ms	18 ms

For the Sobel Filter, OpenCV was actually not that fast. In fact OpenCV had similar performance to my CPU implementation. Running the Sobel filter on the GPU with cuda was much faster.

Assignment Part 2: Block Matrix Multiply

For this section I was havin a lot of trouble getting the cuda kernel to work. I ended up finding a youtube video discussing one way to do block matrix multiplication with cuda and modifying their solution to work with non-square matricies.

https://www.youtube.com/watch?v=ga2ML1uGr5o&t=2186s

Compile and run instructions

```
cd ./matrix
make
./mm arg1 arg2
```

arg1 = M, arg2 = N.

or after `make` to batch run with various M and N values

```
source mntest.sh
```

Deliverables

1. Final Code

- All code is here: ./matrix

2. Results for various M and N values

Summary of Results

Size	CPU	GPU	speedup	RMSE
16x16	0.12 ms	0.16 ms	0.78x	0.00000
16x256	2.26 ms	0.62 ms	3.66x	0.00000
256x16	1.23 ms	1.41 ms	0.87x	0.00001
256x256	26.95 ms	6.00 ms	4.49x	0.00001
256x512	95.99 ms	23.01 ms	4.17x	0.00001
512x256	49.65 ms	2.93 ms	16.92x	0.00004
512x512	199.51 ms	24.57 ms	8.12x	0.00004
512x1024	764.56 ms	81.43 ms	9.39x	0.00004
1024x512	383.14 ms	33.83 ms	11.33x	0.00011
1024x1024	1553.75 ms	105.12 ms	14.78x	0.00011

Output of `batch_run.sh` Results

```
Running for 16x16
Time CPU = 0.12ms, Time GPU = 0.16ms, Speedup = 0.78x, RMSE = 0.00000
------------------------------------------------------------------------

Running for 16x256
Time CPU = 2.26ms, Time GPU = 0.62ms, Speedup = 3.66x, RMSE = 0.00000
------------------------------------------------------------------------

Running for 256x16
Time CPU = 1.23ms, Time GPU = 1.41ms, Speedup = 0.87x, RMSE = 0.00001
------------------------------------------------------------------------

Running for 256x256
Time CPU = 26.95ms, Time GPU = 6.00ms, Speedup = 4.49x, RMSE = 0.00001
```

```
--------------------------------------------------------------------

Running for 256x512
Time CPU = 95.99ms, Time GPU = 23.01ms, Speedup = 4.17x, RMSE = 0.00001
--------------------------------------------------------------------

Running for 512x256
Time CPU = 49.65ms, Time GPU = 2.93ms, Speedup = 16.92x, RMSE = 0.00004
--------------------------------------------------------------------

Running for 512x512
Time CPU = 199.51ms, Time GPU = 24.57ms, Speedup = 8.12x, RMSE = 0.00004
--------------------------------------------------------------------

Running for 512x1024
Time CPU = 764.56ms, Time GPU = 81.43ms, Speedup = 9.39x, RMSE = 0.00004
--------------------------------------------------------------------

Running for 1024x512
Time CPU = 383.14ms, Time GPU = 33.83ms, Speedup = 11.33x, RMSE = 0.00011
--------------------------------------------------------------------

Running for 1024x1024
Time CPU = 1553.75ms, Time GPU = 105.12ms, Speedup = 14.78x, RMSE = 0.00011
--------------------------------------------------------------------
```

3. optimal block/grid layout for 512x512 matrices

Block Size	CPU	GPU	speedup	RMSE
4x4	194.09 ms	143.89 ms	1.35x	0.00004
8x8	192.62 ms	28.96 ms	6.65x	0.00004
16x16	184.36 ms	25.59 ms	7.20x	0.00004
32x32	199.07 ms	00.10 ms	NA	128.3565
64x64	187.97 ms	00.85 ms	NA	128.2251

For a 512x512 it looks like using 8x8 or 16x16 block sizes gives similar performance. Using 4x4 was noticeably slower, but still produced the correct result. Using 32x32 or 64x64 resulted in an incorrect GPU solution. I think I might be running into the limit of how much the GPU block cache can store in shared memory for the larger block sizes.