PIYALI DAS

5441 Lab4 Project Report

INTRODUCTION

In this lab we are supposed to contrast the performance of the serial program on CPU with an equivalent parallel program using the CUDA based code for GPU. We are trying to see how the performance of certain problems can be improved using the GPU. In particular, we are looking at two different problems for the same and will be reporting the results below.
The following sections have the timing results and summary of the lab.

RUNNING THE PROGRAM

I am using a single makefile to compile both the programs of Part 1 and part 2. To request node with GPU in the OSC cluster, I use the command:

qsub -I -l walltime=0:29:00 -l nodes=1:gpus=1 .

Also to load cuda we need “module load cuda”. To individually compile and run for Part 1:
Compiling : nvcc -O -o lab4p1 das_piyali_lab4p1.cu
Run: ./lab4p1

For Part2: nvcc das_piyali_lab4p2.cu bmp_reader.o -o lab4p2
Run: ./lab4p2 image01.bmp out_serial.bmp out_cuda.bmp

RESULTS

- ***PART 1***

For the 1st problem I used 2 dimensional blocks with 2 dimensional threads in each of the blocks. For this problem, after trying out a few combinations, I chose 8 * 8 threads for a block size of 32 * 32. This results in 64 threads per block in total and the best performance among the different configurations. I also used the loop unrolling technique for this problem as it somewhat improves the performance.

Note: I changed the problem size to deal with a 256 * 256 matrix so that the execution times are reasonable because the execution time on the CPU with the original problem

size was around 45 minutes. Later I also tried with a 1024 *1024 matrix and 2048 * 2048 matrix with the same setup.

TIMING INFORMATION
Time taken for 256 * 256 matrix on CPU (sec) = 0.19674, Performance (GFlops/sec) = 0.17055
Time taken for 256 * 256 matrix transpose multiply on GPU (sec) = 0.00651, Performance (GFlops/sec) = 5.15749

Matrix Size	CPU (sec)	CPU Performance (GFlops/sec)	GPU (sec)	GPU Performance (GFlops/sec)
256 * 256	0.19674	0.17055	0.00651	5.15749
1024 *1024	8.33850	0.25754	0.39645	5.41679
2048 *2048	217.46539	0.07900	3.18025	5.40204

So the GPU was approximately 20 times faster than the CPU implementation for problem 1.

- ***PART 2***

I have copied all the test images into my local OSC directory. The results and timing information for the serial and parallel (cuda) code of the Sobel operator is included here:

Test Image (.bmp)	Dimension	Threshold		Timing CPU(sec)		Performance (GFlops/sec)	
		Serial	Cuda	Serial	Cuda	Serial	Cuda
image01	1080*1920	42	42	3.79186	0.45229	0.05462	0.45780
image02	1600*2560	32	32	5.37935	0.61245	0.07684	0.66811
image03	2160*3840	39	38	13.9913 1	1.64188	0.05924	0.50481

image04	1600*2560	156	156	28.31025	3.54959	0.01445	0.11528
image05	1080*1920	54	54	4.90120	0.59193	0.04224	0.34980
image06	1125*1500	16	16	1.21989	0.13614	0.13822	1.23764
image07	1080*1920	36	36	2.99996	0.34292	0.06905	0.60382
image08	1200*1600	105	105	8.83432	1.06659	0.02171	0.17975
image09	1080*1920	40	40	3.66949	0.44271	0.05649	0.46771
coins	246*300	49	44	0.22052	0.01625	0.03325	0.45599

- Yes there was a performance increase of approximately 9 times in this problem by using GPU.
- For the 2nd problem I used threads in x direction and y direction and allocated threads to be multiple of the total number of image pixels.
- Threads per block is a function of x, y coordinates and the number of blocks is represented as number of pixels in image height/thread per block in the x direction and similarly pixels representing image width/thread per block in the y direction