CS499: Introduction to Parallel Programming

Assignment 7: Introduction to General Purpose Computing on Graphics Processing Units

Due: see BBLearn

Preliminaries

You are expected to do your own work on all homework assignments. You may (and are encouraged to) engage in discussions with your classmates regarding the assignments, but specific details of a solution, including the solution itself, must always be your own work. See the academic dishonesty policy in the course syllabus.

Submission Instructions

You should turn in an electronic archive (.zip, .tar., .tgz, etc.). The archive must contain a single top-level directory called CS499_aX_NAME, where "NAME" is your NAU username and "X" is the assignment number (e.g., CS499_a1_mg1234). Inside that directory you should have all your code (no binaries and other compiled code) and requested files, named exactly as specified in the questions below. In the event that I cannot compile your code, you may (or may not) receive an e-mail from me shortly after the assignment deadline. This depends on the nature of the compilation errors. If you do not promptly reply to the e-mail then you may receive a 0 on some of the programming components of the assignment. Because I want to avoid compilation problems, it is crucial that you use the software described in Assignment 1. Assignments need to be turned in via BBLearn.

Turn in a single pdf document that outlines the results of each question. For instance, screenshots that show you achieved the desired program output and a brief text explanation. If you were not able to solve a problem, please provide a brief write up (and screenshots as appropriate) that describes what you tried and why you think it does not work (or why you think it should work). You must provide this brief write up for each programming question in the assignment.

This pdf should be independent of the source code archive, but feel free to include a copy in the top level of that archive as well.

Validation of all Programming Questions

You are responsible for determining the correctness of your programs. There is no testing script for this assignment.

Use of Monsoon

You must use Monsoon when executing all GPU programs. Part of this assignment is demonstrating familiarity with the cluster environment. If you have an Nvidia GPU, you are welcome to use it for development/debugging. But, the results reported in the assignment must be run on one of the GPUs on Monsoon.

Question 1: Introduction to Monsoon and GPU Preliminaries [3 Points]

1. Log into Monsoon and run a GPU job using a job script that was presented in the Monsoon onboarding workshop. The script should have the line: `#SBATCH --qos=gpu_class`. Modify the job script to sleep for 1 minute and then run the program `nvidia-smi`. Report: (*i*) which compute node your job ran on; and (*ii*) what GPUs are installed in the compute node.

2. There has been a rapid evolution in GPU hardware. Consequently, there are features that are only available on particular GPU architectures, and the compiler must know which architecture to compile your CUDA code for. Using the information in the question above, report the CUDA compute capability associated with the GPU. You can look up the compute capability version on Wikipedia: https://en.wikipedia.org/wiki/CUDA.

3. Report the single line that you will use to compile your programs for this assignment. The line will look like the following:
`nvcc -arch=compute_X -code=sm_X -lcuda main.cu -o main`,
where you must determine the value for "X" using the compute capability above. A valuable resource is the the CUDA Toolkit Documentation: https://docs.nvidia.com/cuda/index.html. Report the single line as an example in your pdf.

Submission:
Be sure to include the SLURM script that you used to submit your job in part 1 above.

Question 2: Vector Add [4 Points]

1. Run the vector add program (`vector_add_starter.cu`) on Monsoon with the following values of N: N=100, N=1000, N=1000000, N=100000000, N=500000000. In a table, report the following for each value of N: (*i*) the total time to execute the CPU sequential version; (*ii*) the total time to run the GPU version; (*iii*) the total time to transfer the data to and from the GPU; and (*iv*) the total time to perform the computation in the GPU kernel. Is the performance of the program good or bad? Explain.

   Hint: `cudaDeviceSynchronize()` will come in handy. We saw timing and using `cudaDeviceSynchronize()` in class. You are welcome to use any timing method you wish, as long as you report the time in seconds. To time with `omp_get_wtime()`, you must compile with `-Xcompiler -fopenmp`, where `-Xcompiler` forwards `-fopenmp` to the host compiler. Also, be sure to run a "warm up" kernel first before you make your time measurements to ensure that you exclude initial GPU-related overheads (driver initialization overhead, start up time related to power saving settings, etc.).

Submission:
Use the starter code `vector_add_starter.cu` as a starting point. Download it, rename it `question2_NAME.cu`, complete the question, and turn in an archive with this file. Report how you compiled your program (the single line from Question 1, modified with the file name).

Question 3: Computing the Distances Between Points [8 Points]

In Assignment 6, we computed the distances between points in parallel. Using the same problem formulation in Assignment 6 ($\epsilon = 5.0$, $\epsilon = 10.0$) complete the following questions. For each question, you will execute the program over 3 time trials for N=100, N=1000, N=10000, and N=100000 (24 executions in total). Furthermore, for each GPU execution, you will collect and report two different time measurements. You will report: (*i*) the total time, which includes transferring the data between the CPU and GPU; and (*ii*) the time to only execute the GPU kernel. Like assignment 6, do not include the time needed to generate the data. Use the following starter code for assignment 6, which has been slightly modified: `point_epsilon_starter.cu`. It includes the .cu extension and a definition of the GPU block size: `#define BLOCKSIZE 1024`. You may use this block size (or change it), and you can use it in your GPU kernels. Hint: it will be needed in the bonus question, as the compiler must know the block size at compile time when allocating shared memory *in the kernel*.

1. Implement the $O(n^2)$ brute force solution on the GPU that only uses global memory (do not use shared memory), where each thread is assigned a single point to compare to all other points. Compare

the times to the parallel multi-core brute force method (compiled with -O3) you implemented in Assignment 6, Question 1, for the values of N above, and report the number of CPU cores/threads used. Which is faster, the GPU or CPU implementation? Is the GPU good for small or large N? What is interesting about the response times across values of N on the GPU? Reason about the performance across N on the GPU. [3 Points]

2. Implement a more efficient algorithm than the $O(n^2)$ brute force solution on the GPU. The program does not need to be as (potentially) elaborate as your algorithmic transformation in Assignment 6. How does your new algorithm work? When developing your algorithm, what worked well and what didn't work well? Compare performance with the global memory implementation above (Part 1). Is your implementation faster? Note that your algorithm may be more efficient when N>100000. Feel free to use a larger N, if you believe your approach will outperform the brute force algorithm at larger input sizes. Furthermore, you are not restricted to using global memory, you may also use shared memory. [5 Points]

Submission:
For each of the programs you implement above, name your programs as follows: `question3_pX_NAME.cu`, where X is 1 or 2, corresponding to the two parts above. Complete the questions, and turn in an archive with these files. Report how you compiled your programs (the single line from Question 1, modified with the file name).

Bonus [1 Point]

Implement the $O(n^2)$ brute force solution on the GPU, but exploit shared memory. You must tile the data from global to shared memory, where each thread in the block manually pages a single element into shared memory, and then all threads use the data in shared memory when computing the distances between points. In class, we saw an example of using shared memory to compute the biggest number. In this case, you will need to create an array of shared memory of BLOCKSIZE elements. Compare the shared memory implementation to the global memory implementation above (Question 3, Part 1). Which is faster? Why? You may compare to the GPU kernel only response times, since data transfer times will be the same in both implementations.
Submission:
Complete the question, and turn in an archive with this file, named `questionBonus_NAME.cu`. Report how you compiled your program (the single line from Question 1, modified with the file name).