CS 698L Semester 2019–2020-I: Assignment 5

6th November 2019

Due Your assignment is due by Nov 20 2019 11:59 PM IST.

General Policies

- You should do this assignment ALONE.
- Do not plagiarize or turn in solutions from other sources. You will be PENALIZED if caught.
- We MAY check your submission(s) with plagiarism checkers.

Submission

- Write your programs in C, C++, and CUDA.
- Submission will be through Canvas. Submit a compressed file with name "<roll-no>.tar.gz". The compressed file should have the following structure.

```
roll-no
--report.pdf
--<problem1-dir>
----source-files
--<problem2-dir>
----source-files
```

  Include your source files. The "report.pdf" file should include your name and roll number, and results and explanations that have been asked for in the following problems. In addition to the results, also include the exact compilation instructions for each programming problem, for example, `nvcc problem1.cu`.

  You are encouraged to use the LaTeX typesetting system for generating the PDF file.

- For this assignment, you SHOULD use a desktop and the GPU1 server in the CSE department.
- For late submissions, email your submission to the instructor.
- Submitting your assignments late will mean losing points automatically. You will lose 10% for each day that you miss, for up to three days.

Evaluation

- Please write your code such that the EXACT output format (if specified) is respected.
- We will compile and evaluate your implementations on the lab machines and GPU1 server.
- The evaluators are not expected to fix compilation issues.
- We will evaluate the implementations with our OWN inputs and test cases, so remember to test thoroughly.
- Best submissions (i.e., in terms of performance and suggested optimizations) for Problems 3 and 4 will get 20% bonus based on the problem's worth, at the discretion of the evaluators.

Problem 1

[30 points]

Implement a parallel version of "find max" function for a given array of numbers, using the parallel_reduce template function in Intel TBB. Report the performance of the serial version and the parallel version. Assume the max size of the input array to be limited to 2^{26}.

Report the index of the first occurrence of the max value in case there are duplicates.

Problem 2

[30 points]

(a) Create a parallel kernel using CUDA for the following code. Obviously, make sure that the result is correct.

```
#define SIZE 4096
double F[SIZE][SIZE];

for (int k=0; k<100; k++)
  for (int i=1; i<SIZE; i++)
    for (int j=0; j<SIZE-1; j++)
      F[i][j+1] = F[i-1][j+1]+F[i][j+1];
```

(b) Now try and run the same kernel with SIZE equal to 4097. You are free to try all valid optimizations to to achieve high performance for this kernel.

Report the performance of the two kernels, and explain performance difference if any.

Problem 3

[40 points]

Implement an *optimized* CUDA kernel for multiplying the transpose of a square matrix A with itself (i.e., A^TA). Assume the size is a power of two. Sequential code for the computation is given below.

```
#define SIZE 4096
double F[4096][4096];
// C = (A^T)A
for (int i=0;i<SIZE; i++)
  for (int j=0;j<SIZE; j++)
    for (int k=0;k<SIZE; k++)
      C[i][j] += A[k][i]*A[k][j];
```

Report the performance of the serial version and the parallel version. You are free to try all valid optimizations to to achieve high performance for this kernel.

Problem 4

[50 points]

Implement two versions of matrix multiplication with CUDA: a naïve version and an optimized kernel. You are free to use any valid trick like blocking/tiling and loop unrolling for the second optimized kernel. Compare the correctness of your results with the serial version, and report speedups with the two CUDA implementations. Strive for getting as much speedup as possible with the second optimized kernel.

For this problem, assume that the matrices are square and the size is a power of two. Initialize the matrix with random contents.

Your report should investigate and describe how each of the following factors influence performance of the kernel:

- The size of matrices to be multiplied, using the following sizes: 512, 1024, 2048, and 4096.
- The size of the block/tile computed by each thread block. Experiment with block sizes of say 16, 64, 128, and 256.
- Any other optimizations that you tried for the second CUDA kernel. For example, you may use #pragma unroll to unroll your loop or use manual unrolling.