CSC 410 Exam 1
Daniel Albl

Program Description:

This program performs Floyd's "All-Pairs-Shorted-Path" algorithm for a graph of variable size N, with random non-negative edge weights, using parallel programming. It is implemented twice by two separate main functions, one using OpenMP, and the other using CUDA.

Algorithms Used:

The graph is represented by an NxN matrix A where A[i,j] is the weight of the directed edge from i to j. This matrix is then transformed so that A[i,j] is the distance of the *shortest path* from i to j. Both implementations use a loop over a variable k from 0 to N-1, where the kth iteration calculates the shortest path between all pairs of nodes, given that only nodes <= k can be used as intermediate nodes in the path. Although this loop must be done sequentially, all N^2 shortest paths can be calculated in parallel within each iteration.

If there is no edge between two nodes, the weight is assigned to the programmer's version of infinity, INT_MAX, found in the <climits> library. The <random> library is also used for generating the random matrix.

Functions Used:

There are basic utility functions for randomly initializing and freeing the matrix. To make memory allocation easier and potentially reduce cache misses, the matrix is represented as a 1-D array in memory. For convenience there is a function "at" to calculate the index given an i and j. Both the OpenMP and CUDA implementations have a function for running Floyd's algorithm, called "floyd," and a function for testing the correctness of this algorithm on some hard-coded test cases. The OpenMP version also uses a helper function to handle the overflow caused by adding a number to INT_MAX.

Testing:

I was given two examples of Floyd's algorithm that gave the input and output matrices for a graph with 6 nodes and a graph with 10 nodes. These were hard-coded as global "const int" arrays. The input arrays were copied to the matrix A using <cstring>'s "memcpy" and compared to the output arrays using "memcmp". These testing functions are still in the files provided.

Performance Data:

	Sequential Time (s)	Parellel Time (s)	Processors	Speedup	Efficiency	Sequential Fraction
OMP	11.54	2.89	8	3.993	0.4991	0.1434
CUDA	11.54	0.04279	1024	269.7	0.2634	0.002734

Analysis:

With 128 times as many processors, the CUDA implementation unsurprisingly had a much larger speed up. The efficiency however was higher for OpenMP, which I would accredit to the CPU being faster than the GPU for a single operation, and the purely sequential run-time being measured on the CPU. What I initially found confusing was that despite having higher efficiency, OpenMP had a much larger sequential fraction. My guess is that the overhead of creating threads on the CPU is much larger than on the GPU, giving OpenMP more sequential processing to do. The fact that the CPU is faster per operation does not effect the ratio of sequential to parallel, as it would scale them both down

equally, but it would result in higher efficiency. As the number of processors increases, the sequential fraction becomes more important, making CUDA the more scalable option. Also as N increases, so does the potential speedup since the sequential program is O(N^3) and the parallel program approaches O(N) as you add more processors.

Files Included:

global.h – code used by both implementations
omp.cpp – OpenMP implementation
cuda.cu – CUDA implementation
Makefile

How to run:

$ make
$ ./omp <number of nodes in graph>
$ ./cuda <number of nodes in graph>