COMP5112 Parallel Programming

Assignment 3: CUDA Programming

Due on 5pm on 30th. Nov, 2017

Instructions

- This assignment counts for 20 points.
- This is an individual assignment. You can discuss with others and search online resources but your submission should be your own code.
- Add your name, student id and email as the first line of comments.
- Submit your assignment through Canvas before the deadline.
- Your submission will be compiled and tested on CS lab2 (room 4214) machines.
- **No late submissions will be accepted!**

Assignment Description

Bellman-Ford algorithm is a well-known solution to "the single-source shortest path(SSSP)" problem. It is slower than Dijkstra's algorithm, but more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers.

The input graph *G(V, E)* for this assignment is connected, directed and may contain negative weights. The algorithm finds a shortest path from a specified vertex (the '*source vertex*') to every other vertex in the graph. If there is a negative cycle (a cycle on which the sum of weights is negative) in the graph, there will be no shortest path. In this case, the algorithm will find no result.

In this assignment, you will implement an **CUDA version** of Bellman-Ford algorithm.

The input file will be in following format:

1. The first line is an integer `N`, the number of vertices in the input graph.
2. The following lines are an `N*N` adjacency matrix `mat`, one line per row. The entry in row `v` and column `w`, `mat[v][w],` is the distance (weight) from vertex `v` to vertex `w`. All distances are integers. If there is no edge joining vertex `v` and `w`, `mat[v][w]` will be `1000000` to represent infinity.

The vertex labels are non-negative, consecutive integers, for an input graph with `N` vertices, the vertices will be labeled by `0, 1, 2, …, N-1`. We always use vertex `0` as the source vertex.
The output file of your program consists the distances from vertex `0` to all vertices, in

the increasing order of the vertex label (vertex 0, 1, 2, … and so on), one distance per line. If there are at least one negative cycle (the sum of the weights of the cycle is negative in the graph), your program will set variable `has_negative_cycle` to `true` and print "`FOUND NEGATIVE CYCLE!`" as there will be no shortest path.

Here are two examples input/output for your reference:

Example 1:
Input:

```
3
0 3 2
1000000 0 -2
1000000 2 0
```

Output:

```
0
3
1
```

Example 2:
Input:

```
4
0 100 100 100
100 0 100 -1
100 -1 0 100
100 100 -1 0
```

Output:

```
FOUND NEGATIVE CYCLE!
```

The code skeleton `cuda_bellman_ford.cpp` is provided. You task is to complete the following function in the code:

void bellman_ford
*(int blocksPerGrid, int threadsPerBlock, int n, int *mat, int *dist, bool *has_negative_cycle)*

The description of the parameters is as follows:

Parameter	Description
int blocksPerGrid	Number of blocks for each grid.
int threadsPerBlock	Number of threads for each block.
int n	Number of vertices.
*int *mat*	Adjacency matrix (stored in **one dimension**), N * N elements
*int *dist*	The result array storing the final distance from the source for each vertex, N elements
*bool *has_negative_cycle*	set it to **true** if there is negative weight cycle. Otherwise, set it to **false**

The element `mat[v * N + w]` stores distance(weight) from vertex `v` to vertex `w`.

Note 1: The sequential algorithm of Bellman-Ford is provided for your reference. Your parallel version can follow the same logic flow of the sequential version, but you will need to parallelize it in CUDA.

Note 2: You can add helper/kernel functions and variables as you wish, but keep the existing code skeleton unchanged.

Note 3: We will use different input files, possibly with negative weights and cycles and specify different numbers of *blocksPerGrid*/*threadsPerBlock* to test your program (4<=*blocksPerGrid*<=32, 32<=*threadsPerBlock*<=1024 & *threadsPerBlock* is power of 2).

Note 4: The running time and speedup of your program will be considered in grading.