CS6023: GPU Programming

Assignment 2 (7 marks)

Submission deadline: Mar 7, 2021, 23:55 on Moodle

1 Problem Statement

We are given a directed, unweighted graph as input in which each node of the graph has an initial local value associated with it. Each node of the graph is interested in sending some particular value to all of its neighbors. Based on the values received from sender nodes, each receiver node updates its local value. In other words, each node updates the local values of all of its neighboring nodes.

Given the input graph, write the efficient and data-race free GPU code to implement the above functionality such that memory accesses are coalesced while accessing the graph and other related data structures.

2 Input format with example

Example input:

```
4 5
1 2
2 3
1 3
1 4
3 4
5 3 4 2
4
0 1 2 1 3
1 10 12 23 56
2 1 1 1 1
3
```

The first line contains two space-separated integers denoting the number of nodes(v) and directed edges(e) in the input graph. Next, 'e' lines have two space-separated integers representing the source and destinations nodes for a particular edge, respectively. The next line contains the 'v' space-separated integers denoting the initial local values associated with each of the nodes. The next line contains a single integer 'q' denoting the number of times each node operates on its neighbors to update its local value. In each next 'q' line first integer represents the operation that is to be done (0: addition, 1:minimum, 2:maximum, 3:enumerate), and the rest 'v' integers denote the values that a particular node 'v' will propagate to its neighbors. If the operation is enumerate, it will not be followed by the 'v' integers because during enumerate operation, we are interested in printing each node's current local values.

3 Constraints

- $1 \leq v \leq 10000$
- $1 \leq e \leq n*(n-1)/2$
- $1 \leq q \leq 40$

- All the nodes are numbered from 1 to v.
- There are no self-loops in the graph.
- There are no parallel edges in the graph.

4 Output Format

Each enumerate operation prints a single line of 'v' space-separated integers denoting all the nodes' local values in the graph. These local values should be listed sequentially by node number (i.e., 1, 2, 3, ..., v). The other operations(i.e., Addition, Minimum, Maximum) do not print anything but will update each node's local values in the graph.

5 Sample Test-case with Explanation

5.1 Sample Input

```
4 5
1 2
2 3
1 3
1 4
3 4
5 3 4 2
2
0 1 2 1 3
3
```

5.2 Sample Output

```
5 4 7 4
```

5.3 Explanation

The following figure shows the input graph along with the initial local values of each node:

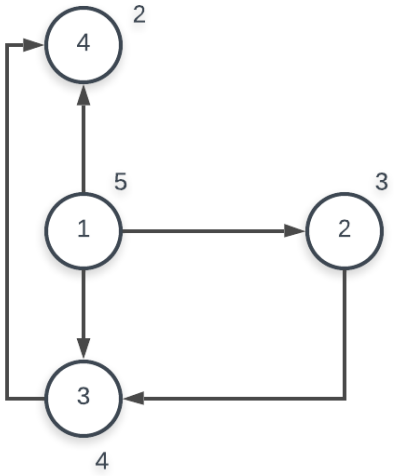

Figure 1: Input Graph

Now, node 1 will add the value 1 to all its neighbor's local values, and hence the current set of local values of all the nodes from 1 to n will be 5 4 5 3, respectively. Similarly, node 2 will add the value 2 to all its neighbor's local values, and hence the current set of local values of all the nodes from 1 to n will be 5 4 7 3, respectively. Similarly, node 3 will add the value 1 to all its neighbor's local values, and hence the current set of local values of all the nodes from 1 to n will be 5 4 7 4 respectively. Node 4 does not have any neighbors; hence, it will not affect any of the nodes' local values. This finishes the execution of the addition operation.
The next enumeration operation will print the current local values of all the nodes from 1 to n, which is 5 4 7 4 respectively.
The operations minimum and maximum will operate in a similar manner as the addition, but instead of adding a value to the neighbor's local value, they will replace the neighbor's local value with the minimum, maximum of neighbor's local value, and the value sent by the node respectively.

6 Points to be noted

- The file 'main.cu' provided by us contains the code, which takes care of file reading, writing, timing the kernel execution, kernel invocations, graph storage, etc. The prototypes of the three kernels are also provided in the same file. You need to implement the three kernels provided in the code.
- Alternatively, you can also write the entire code from scratch on your own if you think that it might give better performance gain than the given code by us. In such a case, you are responsible for timing your kernel's execution time appropriately.
- You can change the names and the signatures of the kernels provided based on your choice.
- **Do not write any print statements inside the kernel.**
- Test your code on large input graphs.

7 Submission guidelines

- You can either use the code provided by us or can write the entire code on your own from scratch based on your choice.
- Compress the file 'main.cu', which contains the implementation of the above-described functionality to ROLL_NUMBER.tar.gz
- Submit only the ROLL_NUMBER.tar.gz on moodle.
- The name of the file should strictly be of the format ROLL_NUMBER.tar.gz
- For example, if your roll number is CS16D019, the name of the file you submit should be CS16D019.tar.gz
- Make sure that the ROLL_NUMBER part of the filename is in upper case.
- Do not upload anything other than the ROLL_NUMBER.tar.gz file.
- After submission, download the file and make sure it was the one you intended to submit.

8 Learning suggestions

- Write a CPU-version of code achieving the same functionality. Time the CPU code and GPU code separately for large graphs and compare the performances.
- Avoid using atomics and other synchronization constructs as much as possible inside the kernels.
- Exploit shared memory as much as possible to gain performance benefits.
- Try reducing thread divergence as much as possible.

9 Bonus Points

- Those submissions which pass all the test-cases and have an execution time of less than half of the whole set of submission's average execution time will get one bonus mark.
- Total time taken by any particular test-case is the sum of time taken by all the kernels in the test-case.
- Total execution time for particular submission is an average of time taken by all the test-cases.