ECSE 420: Parallel Computing

Lab 3: Breadth-First Search

In this lab, you will write code to implement a single iteration of breadth-first search (BFS) with help of shared memory and atomic operations, parallelize it using CUDA, and write a report summarizing your experimental results.

Breadth-First Search (BFS)

Breadth-First Search algorithm is often used to traverse from one node to another node of the graph. In this lab, we are going to simulate one important application of BFS: to help design an integrated circuit by connecting electronic components on the chip. To simplify this procedure: instead of using real electronic components, we are going to use logic gates from lab 1 instead and two nodes are connected as shown in the Figure 1.

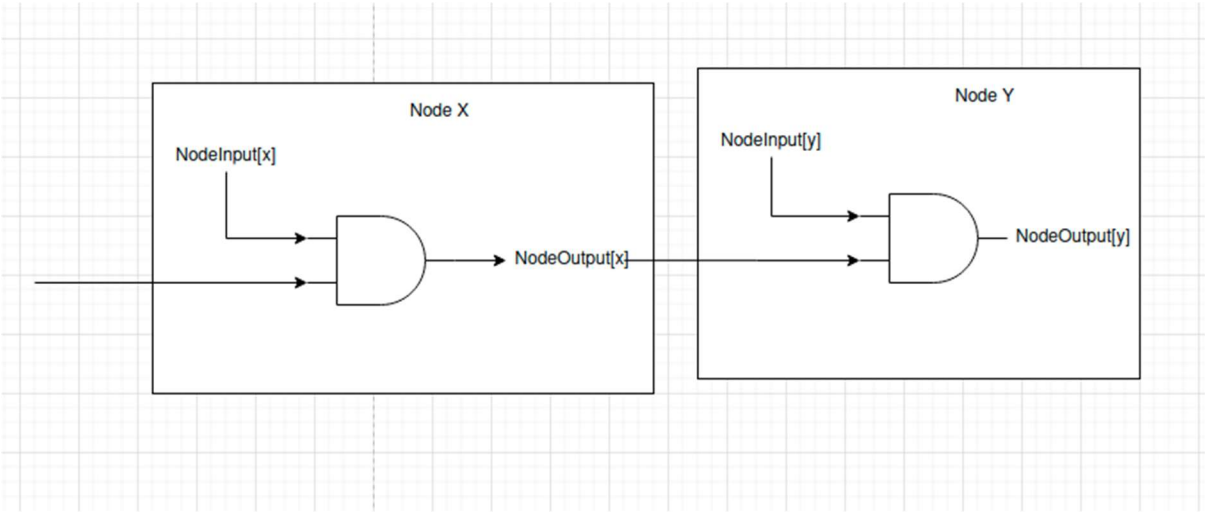

Figure 1 BFS example

Detailed Instruction

Your programs need to take four input files and produce two output files. One output file contains the length of *nodeOutput* (after one iteration) in the first line and all its elements (one line for each element). Another output file contains the length of *numNextLevelNodes in the first line and all of its elements (one line for each element). Two simple programs will be provided to compare the two output files.* **Important: the order of *nextLevelNodes* will be different each time. Please use the correct program to compare the results.**

For each step below, you should verify and benchmark your code against our test case provided and report the execution time obtained.

1. Write code that implements one iteration of BFS sequentially. Your code should take input command like:

   ./sequential <path_to_input_1.raw> <path_to_input_2.raw> <path_to_input_3.raw> <path_to_input_4.raw> <output_nodeOutput_filepath> <output_nextLevelNodes_filepath>

2. Parallelize your code using global queuing with a combination of blockSize (32, 64, 128) and numBlock (10, 25, 35). Your code should take input command like:

   ./global_queuing <path_to_input_1.raw> <path_to_input_2.raw> <path_to_input_3.raw> <path_to_input_4.raw> <output_nodeOutput_filepath> <output_nextLevelNodes_filepath>

3. Parallelize your code using block queuing (shared memory queuing) with a combination of blockSize (32, 64), numBlock (25, 35), and blockQueueCapacity (32, 64). Your code should take input command like:

   ./global_queuing <numBlock> <blockSize> <sharedQueueSize> <path_to_input_1.raw> <path_to_input_2.raw> <path_to_input_3.raw> <path_to_input_4.raw> <output_nodeOutput_filepath> <output_nextLevelNodes_filepath>

4. Discuss your implementations and results and refer to the architecture on which you run your experiment.

Hints:

1. You only need to have *nextLevelNodes* in the queue. In the global queuing, you could directly add elements into *nextLevelNodes* (global queue).

However, for block queuing, you will need to implement a shared memory which is smaller than the length of *nextLevelNodes* and copy it to the *nextLevelNodes* at the end of the thread.

To further simplify this lab, if the block queue is full before the end, you can add your element directly to the global queue.

2. Use atomic operation whenever the element might be accessed by more than one thread at the same time
3. Use __*syncthreads* to make sure all your threads are on the same stage.

Input data format:

There are four input files and one template file to show you how to load inputs correctly.

input1.raw: contains one integer which represents length of file in the first line

and elements of *nodePtrs* (a list of starting indices of nodes' neighbors in input2.raw file),

described below.

input2.raw: contains one integer which represents length of file in the first line

and elements of *nodeNeighbors* (a list of neighbors for each node)

For example, node has *nodeNeighbors[nodePtrs[node]]* to *nodeNeighbors[nodePtrs[node+1]]* describes the neighbors of a node

input3.raw: contains one integer which represents the length of the file in the first line. Rest of the file is in CSV format: each line represents one node and columns represents:

nodeVisited[node] (0: not visited. 1: visited.),
nodeGate[node] (same definition as Lab1),
nodeInput[node] (0 or 1), and
nodeOutput[node] (-1 if not visited. 0 or 1) respectively.

input4.raw: contains one integer which represents the length of the file in the first line and

elements of *currLevelNodes (nodes that currently in the queue)*

Pseudocode codes:

Sequential:

```
// Loop over all nodes in the current level
for idx = 0..numCurrLevelNodes
        node = currLevelNodes[idx];
        // Loop over all neighbors of the node
        For (nbrIdx = nodePtrs[node]..nodePtrs[node + 1];
                neighbor = nodeNeighbors[nbrIdx];
                // If the neighbor hasn't been visited yet
                if !nodeVisited[neighbor]
                // Mark it and add it to the queue
                nodeVisited[neighbor] = 1;
                nodeOutput[neighbor] = gate_solver(nodeGate[neighbor], nodeOutput[node],
                nodeInput[neighbor]);
                nextLevelNodes[*numNextLevelNodes] = neighbor;
                ++(*numNextLevelNodes);
```

Kernels:

```
__global__ void global_queuing_kernel(...){
//Loop over all nodes in the current level
//Loop over all neighbors of the node
//If the neighbor hasn't been visited yet
//Update node output
//Add it to the global queue
}

__global__void block_queuing_kernel(...){
// Initialize shared memory queue
// Loop over all nodes in the current level
//Loop over all neighbors of the node
//If the neighbor hasn't been visited yet
//Update node output
//Add it to the block queue
//If full, add it to the global queue

//Allocate space for block queue to go into global queue
//Store block queue in global queue
}
```

Report Deliverables: (50 points)

- Measure and report the execution time as required **(30 points)**
- Discuss your parallelization scheme and implementation. **(20 points)**

Submission Instructions:

Each group should submit a single zip file with the filename *Group_<your group number> _Lab_3.zip*. (Ex – Group_09_Lab_3.zip). The zip file should contain the following:

1. **A lab report** answering all the questions in the lab (please use a PDF format). **(50 points)**
   a. Must be named *Group_<your group number>_Lab_2_Report.pdf*. (Ex – Group_03_Lab_2_Report.pdf).
   b. Must have a cover page (Include Group number, members' names and ID).
   c. Must follow the logical order for the lab discussions and provide all the items asked for as mentioned above.
2. **Your own source code**. Add the whole project solution. **(50 points)**

Late Submission Penalty:

- **2% per day** of late submission.
- Lab will be accepted up to **1 week after the deadline** (14% max penalty).