

# ECSE420 Lab3 Report

Liang Zhao 260781081  
Hongshuo Zhou 260792817

November 17 2020

Group 6

# 1 Introduction

Generally in this lab, we are required to implement a single iteration of breadth-first search (BFS) with help of shared memory and atomic operations. We need to write code that simulates sequentially and then parallelize our code using global queuing and block queuing (shared memory queuing).

## 2 Design And Testing

### 2.1 Implementation

For this lab, we split the code into three individual block: *sequential.cu*, *globalqueuing.cu*, *blockqueuing.cu*.

### 2.2 Sequential

Our sequential implementation is to directly loop through the input file node by node, and generates the output from the bfs function.

Below is the execution time table, as required, we only report the execution time of the computation, and exclude reading and writing times. We can observe that the execution time is stable. The average running time is 2.751ms.

Experiment Index	Execution time(ms)
1	2.542
2	2.798
3	2.820
4	2.816
5	2.779

Table 1: sequential execution time

Here is our code snippet of Breadth-First Search search implementation, the function takes input node files, then evaluate the corresponding result by loop through the nodes and their neighbours. If the nodes are not visited, they are marked as visited and added to queue.

```

//////////BFS sequential//////////
clock_t start = clock();

// loop over all nodes in the current level
for (int i = 0; i < numCurrLevelNodes; i++) {

    int node = currLevelNodes_h[i];

    // loop over all neighbors of the current node
    for (int j = nodePtrs_h[node]; j < nodePtrs_h[node+1]; j++) {
        int neighbor = nodeNeighbors_h[j];

        // if the isn't visited yet
        if (!nodeVisited_h[neighbor]) {
            // add it to the queue and mark as visited
            nodeVisited_h[neighbor] = 1;
            nodeOutput_h[neighbor] = evaluate_logic_gate(nodeGate_h[neighbor], nodeOutput_h[node], nodeInput_h[neighbor]);
            nextLevelNodes_h[numNextLevelNodes_h] = neighbor;
            ++numNextLevelNodes_h;
        }
    }
}

```

Figure 1: sequential bfs

The output of our algorithm is tested by using given comparison file, and it successfully pass the test, so our output is correct as it is identical to given solution.

Compare output raw files with solutions




 !./compareNodeOutput nodeOutput\_seq.raw sol\_nodeOutput.raw  
 !./compareNextLevelNodes nextLevelNodes\_seq.raw sol\_nextLevelNodes.raw  
 Total Errors : 0 No errors!

Figure 2: Comparison Test

## 2.3 Parallel

### 2.3.1 Global Queuing

Figure 3 is our code snippet of global queuing, the code would allocate cuda memories for corresponding categories of nodes. The categories of nodes are stated by variable names. After that our code would copy data from host to device and pass to execution part to evaluate the result by running Breadth-First Search algorithm.

Figure 4 is the part of the code performed the task of global queuing, it paralleled evaluating the corresponding result by loop through the nodes and their neighbours. If the nodes are not visited, they are marked as visited and added to queue.

```
// allocation
cudaMalloc((void**)&nodePtrs_cuda, 200001 * sizeof(int));
cudaMalloc((void**)&nodeNeighbors_cuda, 50000 * sizeof(int));
cudaMalloc((void**)&nodeVisited_cuda, 200000 * sizeof(int));
cudaMalloc((void**)&nodeGate_cuda, 200000 * sizeof(int));
cudaMalloc((void**)&nodeInput_cuda, 200000 * sizeof(int));
cudaMalloc((void**)&nodeOutput_cuda, 200000 * sizeof(int));
cudaMalloc((void**)&currLevelNodes_cuda, 10000 * sizeof(int));
cudaMallocManaged((void**)&numNextLevelNodes_h, sizeof(int));

*numNextLevelNodes_h = 0;

read_input_one_two_four(&nodePtrs_h, input1);
read_input_one_two_four(&nodeNeighbors_h, input2);
numNodes = read_input_three(&nodeVisited_h, &nodeGate_h, &nodeInput_h, &nodeOutput_h, input3);
numCurrLevelNodes = read_input_one_two_four(&currLevelNodes_h, input4);

// output
int *nextLevelNodes_h = (int*) malloc(numNodes * sizeof(int));
int *nextLevelNodes_cuda;
cudaMalloc((void**)&nextLevelNodes_cuda, numNodes * sizeof(int));

// copy data from host to device
cudaMemcpy(nodePtrs_cuda, nodePtrs_h, 200001 * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(nodeNeighbors_cuda, nodeNeighbors_h, 50000 * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(nodeVisited_cuda, nodeVisited_h, 200000 * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(nodeGate_cuda, nodeGate_h, 200000 * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(nodeInput_cuda, nodeInput_h, 200000 * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(nodeOutput_cuda, nodeOutput_h, 200000 * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(currLevelNodes_cuda, currLevelNodes_h, 10000 * sizeof(int), cudaMemcpyHostToDevice);
```

Figure 3: memory allocation

```
__global__ void global_queuing(int totalThreads, int *nodePtrs_cuda, int *nodeNeighbors_cuda, int *nodeVisited_cuda, int *currLevelNodes_cuda,
int *nodeGate_cuda, int *nodeInput_cuda, int *nodeOutput_cuda, int *nextLevelNodes_cuda, int *numNextLevelNodes, int
int idx = threadIdx.x + blockIdx.x * blockDim.x;
for (; idx < numCurrLevelNodes; idx += totalThreads) {
    int node = currLevelNodes_cuda[idx];
    // Visit all neighbors of the current node
    int nbrIdx;
    for (nbrIdx = nodePtrs_cuda[node]; nbrIdx < nodePtrs_cuda[node + 1]; nbrIdx++){
        int neighbor = nodeNeighbors_cuda[nbrIdx];
        // If the neighbor is not visited yet
        if (!atomicExch(&nodeVisited_cuda[neighbor], 1)){
            // add it to the queue and mark as visited
            nodeOutput_cuda[neighbor] = evaluate_logic_gate(nodeGate_cuda[neighbor], nodeOutput_cuda[node], nodeInput_cuda[neighbor]);
            int index = atomicAdd(&numNextLevelNodes, 1);
            nextLevelNodes_cuda[index] = neighbor;
        }
    }
}
```

Figure 4: global queuing

Below is the execution time table, as required, we only report the execution time of the computation, and exclude reading and writing times. We can observe that the execution time of global queuing is way faster than using sequential algorithm. The average running time is 0.656ms.

Experiment Index	Execution time(ms)
1	0.618
2	0.608
3	0.660
4	0.726
5	0.669

Table 2: global execution time

The output of our algorithm is tested by using given comparison file, and it successfully pass the test, so our output is correct as it is identical to given solution.

Compare output raw files with solutions

```
[ ] !./compareNodeOutput nodeOutput_gq1.raw sol_nodeOutput.raw
!./compareNextLevelNodes nextLevelNodes_gq1.raw sol_nextLevelNodes.raw
```

Total Errors : 0 No errors!

Figure 5: Global Comparison Test

### 2.3.2 Block queuing

For block queuing(shared memory queuing), the logic of bfs search algorithm is same to our code for global queuing. The difference is that the code would initialize shared memory queue before performing node visiting.

Figure 6 is our code snippet of block queuing(shared memory queuing). Block queuing initialize shared memory queue before performing node visiting, it would add the output nodes to block queue first. If block queue is full, then the nodes will be added to the global queue instead. Then the code allocate memories for block queue to go into global queue and store block queue in global queue. As our code shown, *atomicadd()* would updates node

value at specific addresses for queues.

```

__global__ void block_queueing(int* nodePtrs, int* nodeNeighbors, int* data3, int* numCurrLevelNodes,
int size = sharedQueueSize;
extern __shared__ int queue[];

int i = blockIdx.x * blockDim.x + threadIdx.x;
int offset = 0;
if (i < blockSize * numBlock) {
    offset = iter * blockSize * numBlock + i;

    int node = numCurrLevelNodes[offset];
    // Loop over all neighbors of the node
    for (int nbrIdx = nodePtrs[node]; nbrIdx < nodePtrs[node + 1]; nbrIdx++) {
        int neighbor = nodeNeighbors[nbrIdx];
        // If the neighbor hasn't been visited yet
        if (data3[neighbor * 4] == 0) {
            // Mark it and add it to the queue
            data3[neighbor * 4] = 1;

            int gate = data3[neighbor * 4 + 1];
            int input1 = data3[node * 4 + 3];
            int input2 = data3[neighbor * 4 + 2];
            data3[neighbor * 4 + 3] = evaluate_logic_gate(gate, input1, input2);

            int index = atomicAdd(&num, 1);
            if (index < size) {
                int act_index = atomicAdd(&actual_num, 1);
                queue[index] = neighbor;
            } else {
                int glo_index = atomicAdd(&global_num, 1);
                nextLevelNodes[glo_index] = neighbor;
            }
        }
    }
}

```

Figure 6: Block Queuing

Table 3 is the execution time table, as required, we only report the execution time of the computation, and exclude reading and writing times. We can observe that the execution time of block queuing is better than using sequential computation, but slower than global queuing. The average running time is 2.115ms.

Experiment Index	Execution time(ms)
1	2.292
2	2.140
3	1.923
4	2.159
5	2.063

Table 3: global execution time

The output of our algorithm is tested by using given comparison file, and it successfully pass the test, so our output is correct as it is identical to given solution.

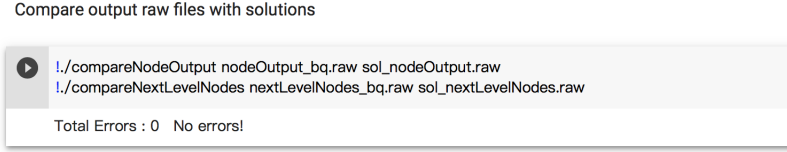


Figure 7: Block Comparison Test

### 3 Conclusion

By summarizing all of the data above, we can conclude that, parallelism certainly improve the execution speed compared to sequential execution. For the two parallel solutions, global queuing and block queuing, global queuing code performed much better. From given material we already knew that the atomic operation acting on block queuing would affect the performance so that the algorithm took longer time to generate desired output. Our lab result strongly support this statement.

Experiment Type	Average Execution time(ms)
Sequential	2.751
Global Queuing	0.656
Block Queuing	2.115

Table 4: Execution Time Comparison