



ECSE420 Parallel Computing

Lab 3 Breadth-First Search

Group 43

Hanwen Wang 260778557

Mai Zeng 260782174

## BFS Sequential

### Code:

In this question, we are asked to implement one iteration of BFS sequentially. We need to read 6 arguments from the terminal, which corresponds to 4 input files and 2 output files. For the node output value, we use logic gates from lab 1. As a result, we implement a function called `gateSolve` to check the gate type and read the output value of the current node and input value of its neighbor node and finally returns the node value calculated. The snippet of code is shown below.

```
int gateSolve(int gate, int output, int input)
{
    if (gate == 0)
    {
        return output & input;
    }
    if (gate == 1)
    {
        return output | input;
    }
    if (gate == 2)
    {
        return !(output & input);
    }
    if (gate == 3)
    {
        return !(output | input);
    }
    if (gate == 4)
    {
        return output ^ input;
    }
    if (gate == 5)
    {
        return !(output ^ input);
    }
    return 0;
}
```

Figure 1: gateSolve function

For the BFS part, we follow the pseudocode shown in the lab manual. The snippet of code is shown below.

```
int BFS(int nodeCurrentLevelLength
, int* nodePtrs, int* nodeNeighbors
, int* nodeVisited, int* nodeGates, int* nodeInput, int* nodeOutput
, int* nodeCurrentLevel
, int* nextLevelNodes)
{
    int numNextLevelNodes = 0;
    for (int i = 0; i < nodeCurrentLevelLength; i++)
    {
        int node = nodeCurrentLevel[i];
        for (int nbrIdx = nodePtrs[node]; nbrIdx < nodePtrs[node + 1]; nbrIdx++)
        {
            int neighbor = nodeNeighbors[nbrIdx];
            if (!nodeVisited[neighbor])
            {
                nodeVisited[neighbor] = 1;
                nodeOutput[neighbor] = gateSolve(nodeGates[neighbor], nodeOutput[node], nodeInput[neighbor]);
                nextLevelNodes[numNextLevelNodes] = neighbor;
                ++numNextLevelNodes;
            }
        }
    }
    return numNextLevelNodes;
}
```

Figure 2: BFS function

First, we loop over all current level nodes (input4.raw) and each line in input4.raw corresponds to the index of line in input1.raw. Then we find its nodePtrs (input1.raw) and define it as nbrldx. Moreover, in input1.raw, value changes by 5 each time it varies, which means that if a node has neighbors, it has exactly 5 neighbors. So nbrldx always loop 5 times in the for loop. In the for loop, we define each node's neighbor as nodeNeighbors[nbrldx] (input2.raw), which will also loop only 5 times as nbrldx. In input2.raw, all values stored are the index of line in input3.raw, as a result, the neighbor we defined is just an index of line in input3.raw. Then we go to input3.raw and check whether that neighbor is visited or not. If it is not visited, we convert it to visited, apply the gateSolve function, add this neighbor to nextLevelNodes output file and update the length of this output file. When all current level nodes are visited, we return the total length of nextLevelNodes output file.

Then in the main function, first we read in 4 input files using the read functions provided and prepare to write contents to 2 output files. After that, we pass in parameters and call BFS function to return the length of nextLevelNodes output file. For nextLevelNodes.raw, we write its length in the first line and then all neighbors' indices in the following content. For nodeOutput.raw, we write the length of file in the first line, which should be equal to length of input3.raw since nodeOutput we defined is just copying all nodeOutput in input3.raw and we are just updating values there, and then all updated nodeOutput values in the following content. The snippet of code is shown below.

```
int main(int argc, char* argv[]) {
    // Variables
    int numNodePtrs;
    int numNodes;
    int* nodePtrs_h;
    int* nodeNeighbors_h;
    int* nodeVisited_h;
    int numTotalNeighbors_h;
    int* currLevelNodes_h;
    int numCurrLevelNodes;
    int numNextLevelNodes_h;
    int* nodeGate_h;
    int* nodeInput_h;
    int* nodeOutput_h;

    FILE* nodeOutputResult;
    FILE* nextLevelNodesResult;

    // read input
    numNodePtrs = read_input_one_two_four(&nodePtrs_h, argv[1]);
    numTotalNeighbors_h = read_input_one_two_four(&nodeNeighbors_h, argv[2]);
    numNodes = read_input_three(&nodeVisited_h, &nodeGate_h, &nodeInput_h, &nodeOutput_h, argv[3]);
    numCurrLevelNodes = read_input_one_two_four(&currLevelNodes_h, argv[4]);

    // write output
    nodeOutputResult = fopen(argv[5], "w");
    nextLevelNodesResult = fopen(argv[6], "w");

    int* nextLevelNodes_h = (int*)malloc(numTotalNeighbors_h * sizeof(int));
    int numNextLevelNodes = BFS(numCurrLevelNodes
        , nodePtrs_h
        , nodeNeighbors_h
        , nodeVisited_h, nodeGate_h, nodeInput_h, nodeOutput_h
        , currLevelNodes_h
        , nextLevelNodes_h);

    int i = 0;
    fprintf(nextLevelNodesResult, "%d\n", numNextLevelNodes);
    while (i < numNextLevelNodes)
    {
        fprintf(nextLevelNodesResult, "%d\n", nextLevelNodes_h[i]);
        i++;
    }

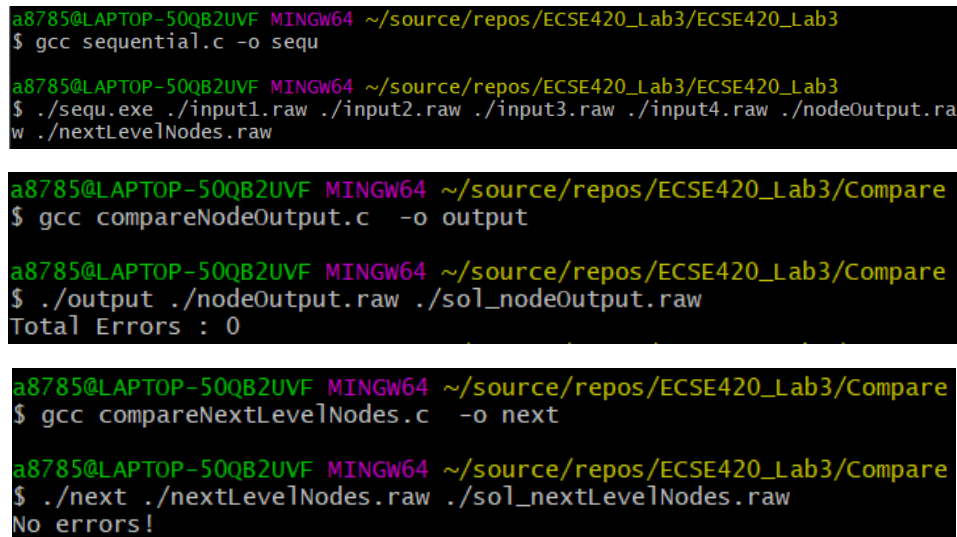
    i = 0;
    fprintf(nodeOutputResult, "%d\n", numNodes);
    while (i < numNodes)
    {
        fprintf(nodeOutputResult, "%d\n", nodeOutput_h[i]);
        i++;
    }

    fclose(nodeOutputResult);
    fclose(nextLevelNodesResult);
}
```

Figure 3: Main function

## Comparison:

We first compile all the c programs in git bash and then compare files we created with solution files. The detailed procedure can be seen in the following figure.



```
a8785@LAPTOP-50QB2UVF MINGW64 ~/source/repos/ECSE420_Lab3/ECSE420_Lab3
$ gcc sequential.c -o sequ

a8785@LAPTOP-50QB2UVF MINGW64 ~/source/repos/ECSE420_Lab3/ECSE420_Lab3
$ ./sequ.exe ./input1.raw ./input2.raw ./input3.raw ./input4.raw ./nodeOutput.raw
./nextLevelNodes.raw

a8785@LAPTOP-50QB2UVF MINGW64 ~/source/repos/ECSE420_Lab3/Compare
$ gcc compareNodeOutput.c -o output

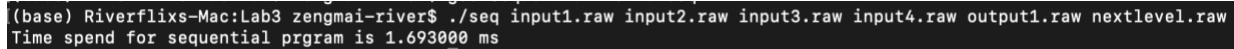
a8785@LAPTOP-50QB2UVF MINGW64 ~/source/repos/ECSE420_Lab3/Compare
$ ./output ./nodeOutput.raw ./sol_nodeOutput.raw
Total Errors : 0

a8785@LAPTOP-50QB2UVF MINGW64 ~/source/repos/ECSE420_Lab3/Compare
$ gcc compareNextLevelNodes.c -o next

a8785@LAPTOP-50QB2UVF MINGW64 ~/source/repos/ECSE420_Lab3/Compare
$ ./next ./nextLevelNodes.raw ./sol_nextLevelNodes.raw
No errors!
```

Figure 4: Compilation and running procedure

As we can see in the results above, there are no errors in both output files, this means that our sequential BFS is working and the time using is about 1.693ms.



```
(base) Riverflx-Mac:Lab3 zengmai-river$ ./seq input1.raw input2.raw input3.raw input4.raw output1.raw nextlevel.raw
Time spend for sequential prgram is 1.693000 ms
```

Figure 5: Time used for sequential part

## Parallel using Global Queuing

### Code:

In this question, we are asked to parallelize our code using global queuing with different combinations of block size and number of blocks. The arguments passed in from terminal is the same as those in the previous question. For the calculation of node output function, it is almost the same as that in the previous question, the only difference is that it is defined as `__device__` because this function is called from the device not the host. For the BFS function running on device, we first set  $i = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$ , which means that this function is running using parallel threads and parallel blocks. Then we declare a for loop, for ( $i < \text{nodeCurrentLevelLength}$ ;  $i += \text{size}$ ), where `nodeCurrentLevelLength` is defined as the length of `input4.raw` and `size` is defined as  $(\text{BLOCK\_SIZE} * \text{NUM\_BLOCK})$ , which also means total number of threads running each time. This function will logically run `size` threads each loop, and when all threads complete running, each of them will add index size and go into the next loop. Inside the for loop, the logic is quite similar to that in the previous question. However, we need to be very careful when updating values. In order to avoid race condition, we use atomic operations to ensure some operations are performed without interference from other threads. We use `atomicCAS` to check whether a neighbor is visited or not and change the value to 1 if it is unvisited at the same time. If that neighbor is unvisited, we calculate its output value and

store it into nodeOutput. Then we use atomicAdd to update the number of nextLevelNodes, and at the same time we use atomicExch to write the neighbor value into its corresponding next position in nextLevel. The snippet of code is shown below.

```

__global__ void global_queueing_bfs_kernel(
    int nodePtrsLength, int nodeNeighborsLength, int nodeVisitedLength, int nodeCurrentLevelLength
    , int* nodePtrs, int* nodeNeighbors
    , int* nodeVisited, int* nodeGates, int* nodeInput, int* nodeOutput
    , int* nodeCurrentLevel
    , int* nextLevel
    , int size
    , int* numNextLevelNodes
)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    for (; i < nodeCurrentLevelLength; i += size) {
        int node = (nodeCurrentLevel[i]);
        for (int nbrIdx = (nodePtrs[node]); nbrIdx < (nodePtrs[node + 1]); nbrIdx++)
        {
            int neighbor = (nodeNeighbors[nbrIdx]);
            syncthreads();
            if (atomicCAS(&nodeVisited[neighbor], 0, 1) == 0)
            {
                nodeOutput[neighbor] = gate_solver(nodeGates[neighbor], nodeOutput[node], nodeInput[neighbor]);
                atomicExch(nextLevel + (atomicAdd(numNextLevelNodes, 1)), neighbor);
            }
        }
    }
}

```

Figure 6: Parallel BFS function

In the host function, we first open each file and store 4 input files' first line as their lengths. Then we define many pointers and use cudaMallocManaged to allocate appropriate space for each of them. This also means that we perform the whole task on unified memory. After that we read each line in four input files, convert the string to int and store into its corresponding pointer we defined earlier. Before we call the kernel function, we start a timer, which is given in Lab0. After the kernel function is done, we stop the timer and print the time elapsed. Afterwards, like what we did in the previous question, we write what we stored in pointers into the two output files with first line showing the length of file and remaining lines showing their corresponding contents. At last, we use cudaFree to free the memory and close all the files. The snippet of code is shown below.

```

void calculation(int BLOCK_SIZE, int NUM_BLOCKS, int argc, char* argv[])
{
    if (argc != 7) {
        printf("The input arguments should follow this format:\n./sequential <path_to_in>\n");
        exit(1);
    }

    /*
    argv[1]: input1
    argv[2]: input2
    argv[3]: input3
    argv[4]: input4
    argv[5]: node output file (output)
    argv[6]: next level nodes (output)
    */

    FILE* input1;
    FILE* input2;
    FILE* input3;
    FILE* input4;
    FILE* nodeOutputResult;
    FILE* nextLevelNodesResult;
    // open files
    input1 = fopen(argv[1], "r");
    if (!input1) {
        perror("File1 open error!\n");
        exit(1);
    }
    input2 = fopen(argv[2], "r");
    if (!input2) {
        perror("File2 open error!\n");
        exit(1);
    }
}

```

```

input3 = fopen(argv[3], "r");
if (!input3) {
    perror("File3 open error!\n");
    exit(1);
}
input4 = fopen(argv[4], "r");
if (!input4) {
    perror("File4 open error!\n");
    exit(1);
}
nodeOutputResult = fopen(argv[5], "w");
nextLevelNodesResult = fopen(argv[6], "w");

// read the first line of the 4 input files and get the length of file
int length1, length2, length3, length4;
fscanf(input1, "%d", &length1);
fscanf(input2, "%d", &length2);
fscanf(input3, "%d", &length3);
fscanf(input4, "%d", &length4);
int* input1Content;
int* input2Content;
int* nodeVisited;
int* nodeGate;
int* nodeInput;
int* nodeOutput;
int* input4Content;
int* nextLevel;
int* numNextLevelNodes;
// Malloc memory for GPU for each array
int size;
cudaMallocManaged((void**)&input1Content, length1 * sizeof(int));
cudaMallocManaged((void**)&input2Content, length2 * sizeof(int));
cudaMallocManaged((void**)&nodeVisited, length3 * sizeof(int));
cudaMallocManaged((void**)&nodeGate, length3 * sizeof(int));
cudaMallocManaged((void**)&nodeInput, length3 * sizeof(int));
cudaMallocManaged((void**)&nodeOutput, length3 * sizeof(int));

cudaMallocManaged((void**)&input4Content, length4 * sizeof(int));
cudaMallocManaged((void**)&nextLevel, length2 * sizeof(int));
cudaMallocManaged((void**)&numNextLevelNodes, 1 * sizeof(int));

// copy the input file 1 content to an array
char* line = (char*)malloc(10);
int i = 0;
while (fgets(line, 10, input1))
{
    if (i == 0)
    {
        i++;
        continue;
    }
    else
    {
        input1Content[i - 1] = atoi(line);
        i++;
    }
}

// copy the input file 2 content to an array
i = 0;
while (fgets(line, 10, input2))
{
    if (i == 0)
    {
        i++;
        continue;
    }
    else
    {
        input2Content[i - 1] = atoi(line);
        i++;
    }
}

// copy the input file 3 content to an array
i = 0;
while (fgets(line, 10, input3))
{
    if (i == 0)
    {
        i++;
        continue;
    }
    else
    {
        nodeVisited[i - 1] = line[0] - 48;
        nodeGate[i - 1] = line[2] - 48;
        nodeInput[i - 1] = line[4] - 48;
        if (line[6] == '-')
        {
            nodeOutput[i - 1] = -1;
        }
        else
        {
            nodeOutput[i - 1] = line[6] - 48;
        }
        i++;
    }
}

// copy the input file 4 content to an array
i = 0;
while (fgets(line, 10, input4))
{
    if (i == 0)
    {
        i++;
        continue;
    }
}

```

```

        else
        {
            input4Content[i - 1] = atoi(line);
            i++;
        }
    }
    *numNextLevelNodes = 0;
    size = (NUM_BLOCKS * BLOCK_SIZE);
    printf("Parallel using blockSize %d and numBlock %d\n", BLOCK_SIZE, NUM_BLOCKS);
    GpuTimer timer;
    timer.Start();
    global_queueing_bfs_kernel << <NUM_BLOCKS, BLOCK_SIZE >> > (length1, length2, length3, length4
        , input1Content
        , input2Content
        , nodeVisited, nodeGate, nodeInput
        , nodeOutput
        , input4Content
        , nextLevel
        , size
        , numNextLevelNodes);
    timer.Stop();
    cudaDeviceSynchronize();
    printf("Time for global queueing: %f ms\n", timer.Elapsed());
    i = 0;
    fprintf(nextLevelNodesResult, "%d\n", *numNextLevelNodes);
    while (i < *numNextLevelNodes)
    {
        fprintf(nextLevelNodesResult, "%d\n", nextLevel[i]);
        i++;
    }
    i = 0;
    fprintf(nodeOutputResult, "%d\n", length3);
    while (i < length3)
    {
        fprintf(nodeOutputResult, "%d\n", nodeOutput[i]);
        i++;
    }
    cudaFree(input1Content);
    cudaFree(input2Content);
    cudaFree(nodeVisited);
    cudaFree(nodeGate);
    cudaFree(nodeInput);
    cudaFree(nodeOutput);
    cudaFree(input4Content);
    cudaFree(nextLevel);
    cudaFree(numNextLevelNodes);

    fclose(input1);
    fclose(input2);
    fclose(input3);
    fclose(input4);
    fclose(nodeOutputResult);
    fclose(nextLevelNodesResult);
}

```

Figure 7: Host function

In the main function, we call the host function by passing in the block size and number of blocks to run with argc and argv. The snippet of code is shown below.

```

int main(int argc, char* argv[]) {
    int BLOCK_SIZE[3] = { 32, 64, 128 };
    int NUM_BLOCKS[3] = { 10, 25, 35 };
    for (int i = 0; i < 3; i++) {
        calculation(BLOCK_SIZE[i], NUM_BLOCKS[i], argc, argv);
    }

    return 0;
}

```

Figure 8: Main function

## Compare:

First we set command arguments as follows in debugging in property of the project.

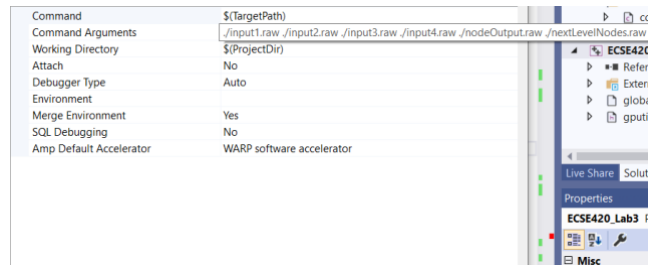


Figure 9: Command arguments setting

Then we launch local windows debugger and get the following result.

```
Parallel using blockSize 32 and numBlock 10  
Time for global queueing: 1.075200 ms  
Parallel using blockSize 64 and numBlock 25  
Time for global queueing: 0.256256 ms  
Parallel using blockSize 128 and numBlock 35  
Time for global queueing: 0.132640 ms
```

Figure 10: Running times shown in debug console

We also got the results running the program from the computer in campus using VPN.



```

mzeng3@156TR4120-05 MINGW64 /p/McGill/ECSE420/Labs/Lab3 (master)
$ ./gq input1.raw input2.raw input3.raw input4.raw outputNode.raw nextlevel.raw
Parallel using blockSize 32 and numBlock 10
Time for global queueing: 0.293600 ms
*numNextLevelNodes = 40101
Parallel using blockSize 32 and numBlock 25
Time for global queueing: 0.133120 ms
*numNextLevelNodes = 40101
Parallel using blockSize 32 and numBlock 35
Time for global queueing: 0.096864 ms
*numNextLevelNodes = 40101
Parallel using blockSize 64 and numBlock 10
Time for global queueing: 0.161760 ms
*numNextLevelNodes = 40101
Parallel using blockSize 64 and numBlock 25
Time for global queueing: 0.084832 ms
*numNextLevelNodes = 40101
Parallel using blockSize 64 and numBlock 35
Time for global queueing: 0.069280 ms
*numNextLevelNodes = 40101
Parallel using blockSize 128 and numBlock 10
Time for global queueing: 0.095136 ms
*numNextLevelNodes = 40101
Parallel using blockSize 128 and numBlock 25
Time for global queueing: 0.069632 ms
*numNextLevelNodes = 40101
Parallel using blockSize 128 and numBlock 35
Time for global queueing: 0.064320 ms
*numNextLevelNodes = 40101

```

Figure 11. Running on the campus computer

Block Size	Number of Block	Time (ms)
32	10	0.293
32	25	0.133
32	35	0.096
64	10	0.161
64	25	0.084
64	35	0.069
128	10	0.095
128	25	0.069
128	35	0.064

Table 1: Time using for each case global queuing

We can see that when block size increases with number of blocks, the time execution of kernel function decreases. This makes sense because with the total number of work unchanged, when applying more threads executing the work, less time should be used. The best performance for global queuing is with block size 128 and number of blocks 35. It makes sense since if we have more threads in one block and more blocks we can do a better parallelism job that is more elements can be process at the same time.

Then we compare the two output files with two solutions provided in git bash. Results are shown below.

```
a8785@LAPTOP-50QB2UVF MINGW64 ~/source/repos/ECSE420_Lab3/Compare
$ ./output ./nodeOutput.raw ./sol_nodeOutput.raw
Total Errors : 0
a8785@LAPTOP-50QB2UVF MINGW64 ~/source/repos/ECSE420_Lab3/Compare
$ ./next ./nextLevelNodes.raw ./sol_nextLevelNodes.raw
No errors!
```

Figure 12: Results shown in git bash

As we can see above, both files have 0 errors. Consequently, our program for global queuing is working as expected.

## Parallel using Block Queuing

In this part, we are asking to parallelize our code using block queuing which is using shared memory queuing. The code snippet shows the kernel function running on GPU. The mainly difference between block queuing and global queuing is that in block queuing we are using shared memory which is shared by the entire block. Once the shared queue in our case it is blockQueueNLs is full, we would add the new elements directly to the global queue.

The \_\_syncthreads function is the barrier for synchronize the whole threads in the block. We specify a check that whether threadIdx.x is equal to 0 this step is to let only one thread add the new elements to the global queue. From line 54 to line 60, we are checking whether the number of elements inside the shared queue is greater than the queue capacity. If the number of elements is bigger than the queue capacity, then we would first let the counter which is also a shared memory variable equal to queue capacity since there is a race condition happens there and then we will use a for loop the upper bound is the value inside the counter which is the queue capacity to add the new elements to the global queue. We are only using the block queue for once every block that why we have that else statement.

```

34 __global__ void block_queueing_bfs_kernel(
35     int nodePtrsLength, int nodeNeighborsLength, int nodeVisitedLength, int nodeCurrentLevelLength
36     , int* nodePtrs, int* nodeNeighbors
37     , int* nodeVisited, int* nodeGates, int* nodeInput, int* nodeOutput
38     , int* nodeCurrentLevel
39     , int* nextLevel
40     , int size
41     , int* numNextLevelNodes
42     , int queueCap
43     , int *num) {
44     int i = blockIdx.x * blockDim.x + threadIdx.x;
45     extern __shared__ int blockQueueNLs[];
46     __shared__ int counter;
47     for(; i < nodeCurrentLevelLength; i += size) {
48         int node = nodeCurrentLevel[i];
49         for(int nbridx = nodePtrs[node]; nbridx < nodePtrs[node+1]; nbridx++) {
50             int neighbor = nodeNeighbors[nbridx];
51             if (atomicCAS(&nodeVisited[neighbor], 0, 1) == 0) {
52                 nodeOutput[neighbor] = gate_solve(nodeGates[neighbor], nodeOutput[node], nodeInput[neighbor]);
53                 int index = atomicAdd(&counter, 1);
54                 if (index < queueCap) {
55                     atomicExch(&(blockQueueNLs[index]), neighbor);
56                 }
57             } else {
58                 atomicExch(&counter, queueCap);
59                 atomicExch(nextLevel+atomicAdd(numNextLevelNodes, 1), neighbor);
60             }
61         }
62     }
63 }
64 __syncthreads();
65
66 if(threadIdx.x == 0) {
67     for(int j = 0; j < counter; j++) {
68         atomicExch(nextLevel+atomicAdd(numNextLevelNodes, 1), blockQueueNLs[j]);
69     }
70 }
71 }
72

```

Figure 13: Code Snippet for block queue function in GPU

The code snippet below shows how we allocate the memory in GPU and it is the same as the previous part which is the global part. The most important part is at line 141 where we need to allocate number of elements in input 3 integers or we can also allocate number of elements in input 2 integers since we are going to traversal all the input 3 elements or we need to traversal all the elements in the input 2 which is the neighbors (5000 elements)

```

122     int* input1Content;
123     int* input2Content;
124     int* nodeVisited;
125     int* nodeGate;
126     int* nodeInput;
127     int* nodeOutput;
128     int* input4Content;
129     int* nextLevel;
130     int* numNextLevelNodes;
131     int* num;
132     // Malloc memory for GPU for each array
133     int size = (blockNum*blockSize);
134     cudaMallocManaged((void**)&input1Content, length1 * sizeof(int));
135     cudaMallocManaged((void**)&input2Content, length2 * sizeof(int));
136     cudaMallocManaged((void**)&nodeVisited, length3 * sizeof(int));
137     cudaMallocManaged((void**)&nodeGate, length3 * sizeof(int));
138     cudaMallocManaged((void**)&nodeInput, length3 * sizeof(int));
139     cudaMallocManaged((void**)&nodeOutput, length3 * sizeof(int));
140     cudaMallocManaged((void**)&input4Content, length4 * sizeof(int));
141     cudaMallocManaged((void**)&nextLevel, length3 * sizeof(int)); // length2 and length3 both can do the job
142     cudaMallocManaged((void**)&numNextLevelNodes, 1*sizeof(int));
143     cudaMallocManaged((void**)&num, 1*sizeof(int));
144

```

Figure 14: Memory allocation for Block Queuing.

The code snippet below shows how we run our gpu function. Note here, shared memory are accessible by multiple threads. To reduce potential bottleneck, shared memory is divided into logical banks. Successive sections of memory are assigned to successive banks. Each bank services only one thread request at a time, multiple simultaneous accesses from different threads to the same bank result in a bank conflict (the accesses are serialized).

Shared memory banks are organized such that successive 32-bit words (4 bytes, one integer size) are assigned to successive banks and the bandwidth is 32 bits per bank per clock cycle and this is the reason why we are using <<<blockNum, blockSize, queueCap\*4>>> below.

```

207     num = 0;
208     timer.Start();
209     block_queuing_bfs_kernel <<<blockNum, blockSize, queueCap*4>>> (length1, length2, length3, length4
210     , input1Content
211     , input2Content
212     , nodeVisited, nodeGate, nodeInput, nodeOutput
213     , input4Content
214     , nextLevel
215     , size
216     , numNextLevelNodes
217     , queueCap
218     , num);
219     timer.Stop();
220     cudaDeviceSynchronize();
221

```

Figure 15: Call the gpu function

The results of this part is showing below, note here we are first writing our code using colab but since we found that sometimes the unix environment is different of the windows environment we decide test our code in both environment. The result below is the windows side using VPN

connecting to the computer in campus. We tested the output using the program provided by the lab and there is no error in our output both next level nodes and also the node output.

We can see that when we increase the shared memory size which is the block queue capability the speed of our program is actually slower than we just use global queue since the access of shared memory is faster.

In table 2, we can also find out that the best performance is when we use 64 threads per block with 35 blocks and a queue capacity of 32 and this is also true when we are running our program in the google Colab. The best performance for using block queue is not better than just use global queue. The reason is that when we use 32 as our block queue, at last when we add these 32 elements into the global queue, we are adding them sequentially and this would need some time to process.

With the blocksize and blocknum increasing, the speed of our program is increasing, and this is already talked in the previous section. With these 2 numbers increasing, more threads are processing the data at the same time and therefore, the speed is faster.

The table below shows each test result.

Number of Blocks	Block Size	Queue Capacity	Time (ms)
32	25	32	0.157
32	25	64	0.167
32	35	32	0.122
32	35	64	0.133
64	25	32	0.101
64	25	64	0.111
64	35	32	0.082
64	35	64	0.093

Table2: Time results

```

azeng38156784120-05 H@ms006 /p/Mcd11/ECSE420/Labs/Lab3 (master)
$ gcc block_queue.cu -o bq
block_queue.cu
Creating library bq.lib and object bq.exp
azeng38156784120-05 H@ms006 /p/Mcd11/ECSE420/Labs/Lab3 (master)
$ ./bq.exe 32 25 32 input1.raw input2.raw input3.raw input4.raw output1.raw nextlevel.raw
start
Time for block queueing: 0.157344 ms with block numbers: 25, threads: 32, queue capacity: 32
azeng38156784120-05 H@ms006 /p/Mcd11/ECSE420/Labs/Lab3 (master)
$ ./bq.exe 32 25 64 input1.raw input2.raw input3.raw input4.raw output1.raw nextlevel.raw
start
Time for block queueing: 0.167936 ms with block numbers: 25, threads: 32, queue capacity: 64
azeng38156784120-05 H@ms006 /p/Mcd11/ECSE420/Labs/Lab3 (master)
$ ./bq.exe 32 35 32 input1.raw input2.raw input3.raw input4.raw output1.raw nextlevel.raw
start
Time for block queueing: 0.122432 ms with block numbers: 35, threads: 32, queue capacity: 32
azeng38156784120-05 H@ms006 /p/Mcd11/ECSE420/Labs/Lab3 (master)
$ ./bq.exe 32 35 64 input1.raw input2.raw input3.raw input4.raw output1.raw nextlevel.raw
start
Time for block queueing: 0.133056 ms with block numbers: 35, threads: 32, queue capacity: 64
azeng38156784120-05 H@ms006 /p/Mcd11/ECSE420/Labs/Lab3 (master)
$ ./bq.exe 64 25 32 input1.raw input2.raw input3.raw input4.raw output1.raw nextlevel.raw
start
Time for block queueing: 0.101376 ms with block numbers: 25, threads: 64, queue capacity: 32
azeng38156784120-05 H@ms006 /p/Mcd11/ECSE420/Labs/Lab3 (master)
$ ./bq.exe 64 25 64 input1.raw input2.raw input3.raw input4.raw output1.raw nextlevel.raw
start
Time for block queueing: 0.111232 ms with block numbers: 25, threads: 64, queue capacity: 64

```

Figure 16: Time results for the block queue method running on campus computer

```

[(base) Riverflixs-Mac:Lab3 zengmai-river$ ./compareNodeOutput ouputbq.raw sol_nodeOutput.raw
Total Errors : 0      (base) Riverflixs-Mac:Lab3 zengmai-river$ 
[(base) Riverflixs-Mac:Lab3 zengmai-river$ ./nextLevelCompare nextlevelbq.raw sol_nextLevelNodes.raw
No errors!

```

Figure 17. Compare Results for block queue method

```

start
Time for block queueing: 0.001829 ms with block numbers: 25, threads: 32, queue capacity: 32
start
Time for block queueing: 0.002139 ms with block numbers: 25, threads: 32, queue capacity: 64
start
Time for block queueing: 0.002103 ms with block numbers: 35, threads: 32, queue capacity: 32
start
Time for block queueing: 0.001875 ms with block numbers: 35, threads: 32, queue capacity: 64
start
Time for block queueing: 0.001727 ms with block numbers: 25, threads: 64, queue capacity: 32
start
Time for block queueing: 0.001918 ms with block numbers: 25, threads: 64, queue capacity: 64
start
Time for block queueing: 0.001893 ms with block numbers: 35, threads: 64, queue capacity: 32
start
Time for block queueing: 0.001888 ms with block numbers: 35, threads: 64, queue capacity: 64

```

Figure 18: Time result running on Google Colab