Module 12 Lab

Breadth-First Search Queue

GPU Teaching Kit – Accelerated Computing

OBJECTIVE

The purpose of this lab is to understand hierarchical queuing in the context of the breadth first search algorithm as an example. You will implement a single iteration of breadth first search that takes a set of nodes in the current level (also called wave-front) as input and outputs the set of nodes belonging to the next level. You will implement two kernels:

- A simple version with global queuing
- An optimized version that uses shared-memory queuing.

INSTRUCTIONS

The graph structure is stored in the following way:

- `numNodes` - the total number of nodes in the graph
- `nodePtrs` - an array of length `numNodes`. Each entry is a pointer into `numNeighbors`, described below.
- `nodeNeighbors` - an array whose length is the total number of neighbors each
  node has. `nodeNeighbors[nodePtrs[node]]` to `nodeNeighbors[nodePtrs[node+1]]` describes the neighbors of node `node`.

The kernels take these structures as inputs, as well as a list of nodes in the current level, for which all of the neighbors must be visited.

- `currLevelNodes` - an array of nodes for which neighbors must be visited
- `numCurrLevelNodes` - the size of the previous array
- `visitedNodes` - an array that describes which nodes have already been visited in the BFS

The kernels will need to produce the following outputs

- `nextLevelNodes` - an array of neighbor nodes.
- `numNextLevelNodes` - the number of neighbors

- `visitedNodes` - the nodes that have been visited by the end of the iteration. Note that this is the same array as the input. This is the output value that WebGPU will compare for correctness of your implementation.

Sequential pseudocode for the kernel is:

```
// 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;
      nextLevelNodes[*numNextLevelNodes] = neighbor;
      ++(*numNextLevelNodes);
```

An empty stub for the kernels is provided. All you need to do is correctly implement the kernel code.

TEST DATASETS

The first three datasets invoke the Global Queue kernel. The second three invoke the Block Queue Kernel

LOCAL SETUP INSTRUCTIONS

The most recent version of source code for this lab along with the build-scripts can be found on the Bitbucket repository. A description on how to use the CMake tool in along with how to build the labs for local development found in the README document in the root of the repository.

The executable generated as a result of compiling the lab can be run using the following command:

```
./BfsQueue_Template -e <expected.raw> \
  -i <input0.raw>,<input1.raw>,<input2.raw>,<input3.raw>,<input4.raw> -t integral_vector
```

where `<expected.raw>` is the expected output, `<input.raw>` is the input dataset. The datasets can be generated using the dataset generator built as part of the compilation process.

QUESTIONS

CODE TEMPLATE

The following code is suggested as a starting point for students. The code handles the import and export as well as the checking of the solution. Students are expected to insert their code is the sections demarcated with `//@@`. Students expected the other code unchanged. The tutorial page describes the functionality of the `wb*` methods.

```
1
2  #include <stdio.h>
3  #include <wb.h>
4
5  #define BLOCK_SIZE 512
6  // Maximum number of elements that can be inserted into a block queue
7  #define BQ_CAPACITY 2048
8
9  #define wbCheck(stmt)                                                     \
10   do {                                                                    \
11     cudaError_t err = stmt;                                               \
12     if (err != cudaSuccess) {                                             \
13       wbLog(ERROR, "Failed to run stmt ", #stmt);                         \
14       return -1;                                                          \
15     }                                                                     \
16   } while (0)
17
18 // Global queuing stub
19 __global__ void gpu_global_queuing_kernel(
20     int *nodePtrs, int *nodeNeighbors, int *nodeVisited,
21     int *currLevelNodes, int *nextLevelNodes,
22     const unsigned int numCurrLevelNodes, int *numNextLevelNodes) {
23
24   //@@ Insert Global Queuing Code Here
25
26   // Loop over all nodes in the curent level
27   // Loop over all neighbors of the node
28   // If the neighbor hasn't been visited yet
29   // Add it to the global queue (already marked in the exchange)
30 }
31
32 // Block queuing stub
33 __global__ void gpu_block_queuing_kernel(
34     int *nodePtrs, int *nodeNeighbors, int *nodeVisited,
35     int *currLevelNodes, int *nextLevelNodes,
36     const unsigned int numCurrLevelNodes, int *numNextLevelNodes) {
37
38   //@@ Insert Block Queuing Code Here
39
40   // Initialize shared memory queue
41
42   // Loop over all nodes in the curent level
43   // Loop over all neighbors of the node
```

```
  // If the neighbor hasn't been visited yet
  // 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
}

// Host function for global queuing invocation
void gpu_global_queuing(int *nodePtrs, int *nodeNeighbors,
                        int *nodeVisited, int *currLevelNodes,
                        int *nextLevelNodes,
                        unsigned int numCurrLevelNodes,
                        int *numNextLevelNodes) {

  const unsigned int numBlocks = 45;
  gpu_global_queuing_kernel<<<numBlocks, BLOCK_SIZE>>>(
      nodePtrs, nodeNeighbors, nodeVisited, currLevelNodes, nextLevelNodes,
      numCurrLevelNodes, numNextLevelNodes);
}

// Host function for block queuing invocation
void gpu_block_queuing(int *nodePtrs, int *nodeNeighbors, int *nodeVisited,
                       int *currLevelNodes, int *nextLevelNodes,
                       unsigned int numCurrLevelNodes,
                       int *numNextLevelNodes) {

  const unsigned int numBlocks = 45;
  gpu_block_queuing_kernel<<<numBlocks, BLOCK_SIZE>>>(
      nodePtrs, nodeNeighbors, nodeVisited, currLevelNodes, nextLevelNodes,
      numCurrLevelNodes, numNextLevelNodes);
}

int main(int argc, char *argv[]) {
  // Variables
  int numNodes;
  int *nodePtrs_h;
  int *nodeNeighbors_h;
  int *nodeVisited_h;
  int numTotalNeighbors_h;
  int *currLevelNodes_h;
  int *nextLevelNodes_h;
  int numCurrLevelNodes;
  int numNextLevelNodes_h;
  int *nodePtrs_d;
  int *nodeNeighbors_d;
  int *nodeVisited_d;
  int *currLevelNodes_d;
  int *nextLevelNodes_d;
  int *numNextLevelNodes_d;

  enum Mode { GPU_GLOBAL_QUEUE = 2, GPU_BLOCK_QUEUE };
```

```
97
98    wbArg_t args = wbArg_read(argc, argv);
99    Mode mode    = (Mode)wbImport_flag(wbArg_getInputFile(args, 0));
100
101   nodePtrs_h =
102       (int *)wbImport(wbArg_getInputFile(args, 1), &numNodes, "Integer");
103   nodeNeighbors_h = (int *)wbImport(wbArg_getInputFile(args, 2),
104                                     &numTotalNeighbors_h, "Integer");
105
106   nodeVisited_h =
107       (int *)wbImport(wbArg_getInputFile(args, 3), &numNodes, "Integer");
108   currLevelNodes_h = (int *)wbImport(wbArg_getInputFile(args, 4),
109                                      &numCurrLevelNodes, "Integer");
110
111   // (do not modify) Datasets should be consistent
112   if (nodePtrs_h[numNodes] != numTotalNeighbors_h) {
113     wbLog(ERROR, "Datasets are inconsistent! Please report this.");
114   }
115
116   // (do not modify) Prepare next level containers (i.e. output variables)
117   numNextLevelNodes_h = 0;
118   nextLevelNodes_h    = (int *)malloc((numNodes) * sizeof(int));
119
120   wbLog(TRACE, "# Modes = ", mode);
121   wbLog(TRACE, "# Nodes = ", numNodes);
122   wbLog(TRACE, "# Total Neighbors = ", numTotalNeighbors_h);
123   wbLog(TRACE, "# Current Level Nodes = ", numCurrLevelNodes);
124
125   // (do not modify) Allocate device variables -------------------------
126
127   wbLog(TRACE, "Allocating device variables...");
128
129   wbCheck(cudaMalloc((void **)&nodePtrs_d, (numNodes + 1) * sizeof(int)));
130   wbCheck(cudaMalloc((void **)&nodeVisited_d, numNodes * sizeof(int)));
131   wbCheck(cudaMalloc((void **)&nodeNeighbors_d,
132                      nodePtrs_h[numNodes] * sizeof(int)));
133   wbCheck(cudaMalloc((void **)&currLevelNodes_d,
134                      numCurrLevelNodes * sizeof(int)));
135   wbCheck(cudaMalloc((void **)&numNextLevelNodes_d, sizeof(int)));
136   wbCheck(
137       cudaMalloc((void **)&nextLevelNodes_d, (numNodes) * sizeof(int)));
138   wbCheck(cudaDeviceSynchronize());
139
140   // (do not modify) Copy host variables to device ----------------------
141
142   wbLog(TRACE, "Copying data from host to device...");
143
144   wbCheck(cudaMemcpy(nodePtrs_d, nodePtrs_h, (numNodes + 1) * sizeof(int),
145                      cudaMemcpyHostToDevice));
146   wbCheck(cudaMemcpy(nodeVisited_d, nodeVisited_h, numNodes * sizeof(int),
147                      cudaMemcpyHostToDevice));
148   wbCheck(cudaMemcpy(nodeNeighbors_d, nodeNeighbors_h,
149                      nodePtrs_h[numNodes] * sizeof(int),
```

```
150                     cudaMemcpyHostToDevice));
151   wbCheck(cudaMemcpy(currLevelNodes_d, currLevelNodes_h,
152                     numCurrLevelNodes * sizeof(int),
153                     cudaMemcpyHostToDevice));
154   wbCheck(cudaMemset(numNextLevelNodes_d, 0, sizeof(int)));
155   wbCheck(cudaDeviceSynchronize());
156
157   // (do not modify) Launch kernel ----------------------------------------
158
159   printf("Launching kernel ");
160
161   if (mode == GPU_GLOBAL_QUEUE) {
162     wbLog(INFO, "(GPU with global queuing)...");
163     gpu_global_queuing(nodePtrs_d, nodeNeighbors_d, nodeVisited_d,
164                        currLevelNodes_d, nextLevelNodes_d,
165                        numCurrLevelNodes, numNextLevelNodes_d);
166     wbCheck(cudaDeviceSynchronize());
167   } else if (mode == GPU_BLOCK_QUEUE) {
168     wbLog(INFO, "(GPU with block and global queuing)...");
169     gpu_block_queuing(nodePtrs_d, nodeNeighbors_d, nodeVisited_d,
170                       currLevelNodes_d, nextLevelNodes_d,
171                       numCurrLevelNodes, numNextLevelNodes_d);
172     wbCheck(cudaDeviceSynchronize());
173   } else {
174     wbLog(ERROR, "Invalid mode!\n");
175     exit(0);
176   }
177
178   // (do not modify) Copy device variables from host ----------------------
179
180   wbLog(INFO, "Copying data from device to host...");
181
182   wbCheck(cudaMemcpy(&numNextLevelNodes_h, numNextLevelNodes_d,
183                     sizeof(int), cudaMemcpyDeviceToHost));
184   wbCheck(cudaMemcpy(nextLevelNodes_h, nextLevelNodes_d,
185                     numNodes * sizeof(int), cudaMemcpyDeviceToHost));
186   wbCheck(cudaMemcpy(nodeVisited_h, nodeVisited_d, numNodes * sizeof(int),
187                     cudaMemcpyDeviceToHost));
188   wbCheck(cudaDeviceSynchronize());
189
190   // (do not modify) Verify correctness
191   // -------------------------------------
192   // Only check that the visited nodes match the reference implementation
193
194   wbSolution(args, nodeVisited_h, numNodes);
195
196   // (do not modify) Free memory
197   // ---------------------------------------------------------
198   free(nodePtrs_h);
199   free(nodeVisited_h);
200   free(nodeNeighbors_h);
201   free(currLevelNodes_h);
202   free(nextLevelNodes_h);
```

```
203   wbCheck(cudaFree(nodePtrs_d));
204   wbCheck(cudaFree(nodeVisited_d));
205   wbCheck(cudaFree(nodeNeighbors_d));
206   wbCheck(cudaFree(currLevelNodes_d));
207   wbCheck(cudaFree(numNextLevelNodes_d));
208   wbCheck(cudaFree(nextLevelNodes_d));
209
210   return 0;
211 }
```

CODE SOLUTION

The following is a possible implementation of the lab. This solution is intended for use only by the teaching staff and should not be distributed to students.

```
1
2  #include <stdio.h>
3  #include <wb.h>
4
5  #define BLOCK_SIZE 512
6  // Maximum number of elements that can be inserted into a block queue
7  #define BQ_CAPACITY 2048
8
9  #define wbCheck(stmt)                                                     \
10   do {                                                                    \
11     cudaError_t err = stmt;                                               \
12     if (err != cudaSuccess) {                                             \
13       wbLog(ERROR, "Failed to run stmt ", #stmt);                         \
14       return -1;                                                          \
15     }                                                                     \
16   } while (0)
17
18 // Global queuing stub
19 __global__ void gpu_global_queuing_kernel(
20     int *nodePtrs, int *nodeNeighbors, int *nodeVisited,
21     int *currLevelNodes, int *nextLevelNodes,
22     const unsigned int numCurrLevelNodes, int *numNextLevelNodes) {
23
24   //@@ Insert Global Queuing Code Here
25
26   const unsigned int tid = blockIdx.x * blockDim.x + threadIdx.x;
27
28   // Loop over all nodes in the curent level
29   for (unsigned int idx = tid; idx < numCurrLevelNodes;
30        idx += gridDim.x * blockDim.x) {
31     const unsigned int node = currLevelNodes[idx];
32     // Loop over all neighbors of the node
33     for (unsigned int nbrIdx = nodePtrs[node]; nbrIdx < nodePtrs[node + 1];
34          ++nbrIdx) {
35       const unsigned int neighbor = nodeNeighbors[nbrIdx];
36       // If the neighbor hasn't been visited yet
```

```
37        const unsigned int visited = atomicExch(&(nodeVisited[neighbor]), 1);
38        if (!visited) {
39          // Add it to the global queue (already marked in the exchange)
40          const unsigned int gQIdx = atomicAdd(numNextLevelNodes, 1);
41          nextLevelNodes[gQIdx]    = neighbor;
42        }
43      }
44    }
45  }
46
47  // Block queuing stub
48  __global__ void gpu_block_queuing_kernel(
49      int *nodePtrs, int *nodeNeighbors, int *nodeVisited,
50      int *currLevelNodes, int *nextLevelNodes,
51      const unsigned int numCurrLevelNodes, int *numNextLevelNodes) {
52
53    //@@ INSERT KERNEL CODE HERE
54
55    const unsigned int tid = blockIdx.x * blockDim.x + threadIdx.x;
56    const unsigned int numCurrLevelNodes_reg = numCurrLevelNodes;
57
58    // Initialize shared memory queue
59    __shared__ int bQueue[BQ_CAPACITY];
60    __shared__ int bQueueCount, gQueueStartIdx;
61    if (threadIdx.x == 0) {
62      bQueueCount = 0;
63    }
64    __syncthreads();
65
66    // Loop over all nodes in the curent level
67    for (unsigned int idx = tid; idx < numCurrLevelNodes_reg;
68         idx += gridDim.x * blockDim.x) {
69      const unsigned int node = currLevelNodes[idx];
70      // Loop over all neighbors of the node
71      for (unsigned int nbrIdx = nodePtrs[node]; nbrIdx < nodePtrs[node + 1];
72           ++nbrIdx) {
73        const unsigned int neighbor = nodeNeighbors[nbrIdx];
74        // If the neighbor hasn't been visited yet
75        const unsigned int visited = atomicExch(&(nodeVisited[neighbor]), 1);
76        if (!visited) {
77          // Add it to the block queue
78          const unsigned int bQueueIdx = atomicAdd(&bQueueCount, 1);
79          if (bQueueIdx < BQ_CAPACITY) {
80            bQueue[bQueueIdx] = neighbor;
81          } else { // If full, add it to the global queue
82            bQueueCount                  = BQ_CAPACITY;
83            const unsigned int gQueueIdx = atomicAdd(numNextLevelNodes, 1);
84            nextLevelNodes[gQueueIdx]    = neighbor;
85          }
86        }
87      }
88    }
89    __syncthreads();
```

```
90
91    // Allocate space for block queue to go into global queue
92    if (threadIdx.x == 0) {
93      gQueueStartIdx = atomicAdd(numNextLevelNodes, bQueueCount);
94    }
95    __syncthreads();
96
97    // Store block queue in global queue
98    for (unsigned int bQueueIdx = threadIdx.x; bQueueIdx < bQueueCount;
99         bQueueIdx += blockDim.x) {
100     nextLevelNodes[gQueueStartIdx + bQueueIdx] = bQueue[bQueueIdx];
101   }
102 }
103
104 // Host function for global queuing invocation
105 void gpu_global_queuing(int *nodePtrs, int *nodeNeighbors,
106                         int *nodeVisited, int *currLevelNodes,
107                         int *nextLevelNodes,
108                         unsigned int numCurrLevelNodes,
109                         int *numNextLevelNodes) {
110
111   const unsigned int numBlocks = 45;
112   gpu_global_queuing_kernel<<<numBlocks, BLOCK_SIZE>>>(
113       nodePtrs, nodeNeighbors, nodeVisited, currLevelNodes, nextLevelNodes,
114       numCurrLevelNodes, numNextLevelNodes);
115 }
116
117 // Host function for block queuing invocation
118 void gpu_block_queuing(int *nodePtrs, int *nodeNeighbors, int *nodeVisited,
119                        int *currLevelNodes, int *nextLevelNodes,
120                        unsigned int numCurrLevelNodes,
121                        int *numNextLevelNodes) {
122
123   const unsigned int numBlocks = 45;
124   gpu_block_queuing_kernel<<<numBlocks, BLOCK_SIZE>>>(
125       nodePtrs, nodeNeighbors, nodeVisited, currLevelNodes, nextLevelNodes,
126       numCurrLevelNodes, numNextLevelNodes);
127 }
128
129 int main(int argc, char *argv[]) {
130   // Variables
131   int numNodes;
132   int *nodePtrs_h;
133   int *nodeNeighbors_h;
134   int *nodeVisited_h;
135   int numTotalNeighbors_h;
136   int *currLevelNodes_h;
137   int *nextLevelNodes_h;
138   int numCurrLevelNodes;
139   int numNextLevelNodes_h;
140   int *nodePtrs_d;
141   int *nodeNeighbors_d;
142   int *nodeVisited_d;
```

```
143     int *currLevelNodes_d;
144     int *nextLevelNodes_d;
145     int *numNextLevelNodes_d;
146
147     enum Mode { GPU_GLOBAL_QUEUE = 2, GPU_BLOCK_QUEUE };
148
149     wbArg_t args = wbArg_read(argc, argv);
150     Mode mode    = (Mode)wbImport_flag(wbArg_getInputFile(args, 0));
151
152     nodePtrs_h =
153         (int *)wbImport(wbArg_getInputFile(args, 1), &numNodes, "Integer");
154     nodeNeighbors_h = (int *)wbImport(wbArg_getInputFile(args, 2),
155                                       &numTotalNeighbors_h, "Integer");
156
157     nodeVisited_h =
158         (int *)wbImport(wbArg_getInputFile(args, 3), &numNodes, "Integer");
159     currLevelNodes_h = (int *)wbImport(wbArg_getInputFile(args, 4),
160                                        &numCurrLevelNodes, "Integer");
161
162     // (do not modify) Datasets should be consistent
163     if (nodePtrs_h[numNodes] != numTotalNeighbors_h) {
164       wbLog(ERROR, "Datasets are inconsistent! Please report this.");
165     }
166
167     // (do not modify) Prepare next level containers (i.e. output variables)
168     numNextLevelNodes_h = 0;
169     nextLevelNodes_h    = (int *)malloc((numNodes) * sizeof(int));
170
171     wbLog(TRACE, "# Modes = ", mode);
172     wbLog(TRACE, "# Nodes = ", numNodes);
173     wbLog(TRACE, "# Total Neighbors = ", numTotalNeighbors_h);
174     wbLog(TRACE, "# Current Level Nodes = ", numCurrLevelNodes);
175
176     // (do not modify) Allocate device variables -------------------------
177
178     wbLog(TRACE, "Allocating device variables...");
179
180     wbCheck(cudaMalloc((void **)&nodePtrs_d, (numNodes + 1) * sizeof(int)));
181     wbCheck(cudaMalloc((void **)&nodeVisited_d, numNodes * sizeof(int)));
182     wbCheck(cudaMalloc((void **)&nodeNeighbors_d,
183                        nodePtrs_h[numNodes] * sizeof(int)));
184     wbCheck(cudaMalloc((void **)&currLevelNodes_d,
185                        numCurrLevelNodes * sizeof(int)));
186     wbCheck(cudaMalloc((void **)&numNextLevelNodes_d, sizeof(int)));
187     wbCheck(
188         cudaMalloc((void **)&nextLevelNodes_d, (numNodes) * sizeof(int)));
189     wbCheck(cudaDeviceSynchronize());
190
191     // (do not modify) Copy host variables to device ----------------------
192
193     wbLog(TRACE, "Copying data from host to device...");
194
195     wbCheck(cudaMemcpy(nodePtrs_d, nodePtrs_h, (numNodes + 1) * sizeof(int),
```

```
196                     cudaMemcpyHostToDevice));
197   wbCheck(cudaMemcpy(nodeVisited_d, nodeVisited_h, numNodes * sizeof(int),
198                     cudaMemcpyHostToDevice));
199   wbCheck(cudaMemcpy(nodeNeighbors_d, nodeNeighbors_h,
200                     nodePtrs_h[numNodes] * sizeof(int),
201                     cudaMemcpyHostToDevice));
202   wbCheck(cudaMemcpy(currLevelNodes_d, currLevelNodes_h,
203                     numCurrLevelNodes * sizeof(int),
204                     cudaMemcpyHostToDevice));
205   wbCheck(cudaMemset(numNextLevelNodes_d, 0, sizeof(int)));
206   wbCheck(cudaDeviceSynchronize());
207
208   // (do not modify) Launch kernel ----------------------------------------
209
210   printf("Launching kernel ");
211
212   if (mode == GPU_GLOBAL_QUEUE) {
213     wbLog(INFO, "(GPU with global queuing)...");
214     gpu_global_queuing(nodePtrs_d, nodeNeighbors_d, nodeVisited_d,
215                        currLevelNodes_d, nextLevelNodes_d,
216                        numCurrLevelNodes, numNextLevelNodes_d);
217     wbCheck(cudaDeviceSynchronize());
218   } else if (mode == GPU_BLOCK_QUEUE) {
219     wbLog(INFO, "(GPU with block and global queuing)...");
220     gpu_block_queuing(nodePtrs_d, nodeNeighbors_d, nodeVisited_d,
221                       currLevelNodes_d, nextLevelNodes_d,
222                       numCurrLevelNodes, numNextLevelNodes_d);
223     wbCheck(cudaDeviceSynchronize());
224   } else {
225     wbLog(ERROR, "Invalid mode!\n");
226     exit(0);
227   }
228
229   // (do not modify) Copy device variables from host ----------------------
230
231   wbLog(INFO, "Copying data from device to host...");
232
233   wbCheck(cudaMemcpy(&numNextLevelNodes_h, numNextLevelNodes_d,
234                     sizeof(int), cudaMemcpyDeviceToHost));
235   wbCheck(cudaMemcpy(nextLevelNodes_h, nextLevelNodes_d,
236                     numNodes * sizeof(int), cudaMemcpyDeviceToHost));
237   wbCheck(cudaMemcpy(nodeVisited_h, nodeVisited_d, numNodes * sizeof(int),
238                     cudaMemcpyDeviceToHost));
239   wbCheck(cudaDeviceSynchronize());
240
241   // (do not modify) Verify correctness
242   // -------------------------------------
243   // Only check that the visited nodes match the reference implementation
244
245   wbSolution(args, nodeVisited_h, numNodes);
246
247   // (do not modify) Free memory
248   // ---------------------------------------------------------
```

```
249     free(nodePtrs_h);
250     free(nodeVisited_h);
251     free(nodeNeighbors_h);
252     free(currLevelNodes_h);
253     free(nextLevelNodes_h);
254     wbCheck(cudaFree(nodePtrs_d));
255     wbCheck(cudaFree(nodeVisited_d));
256     wbCheck(cudaFree(nodeNeighbors_d));
257     wbCheck(cudaFree(currLevelNodes_d));
258     wbCheck(cudaFree(numNextLevelNodes_d));
259     wbCheck(cudaFree(nextLevelNodes_d));
260
261     return 0;
262 }
```