ECSE 420 – Parallel Computing – Fall 2020

Lab 3 – Breadth-First Search

Aleksas Murauskas 260718389

Anne-Julie Côté 260803768

Group 16

1. Sequential implementation of BFS

Our Sequential BFS Implementation begins parsing input, for which it has two helper methods. The methods read_input_one_two_four() and read_input_three() are used to parse their respective raw input files. Both of these inputs begin by opening the passed filename. read_input_one_two_four() first reads the length of input from the input file. An pointer input1 is passed by reference to the method to hold the input file data. A while loop iterates through the input file, assigning the file's value to the array each iteration using fscanf(). The loop concludes when the fscanf() returns a 0, indicating the end of the file has been reached. read_input_three() functions similarly, except it is passed four input pointers and within the while loop that scans the file, each scan assigns a value to each of the input arrays. Both methods conclude with the closure of the file and return the length of the input arrays.

Within main, the first argument is passed to read_input_one_two_four(), which returns the number of node pointers and stores the file data in the array nodePtrs_h. The method is called again to read the second argument's file , in order to find the number of total Neighbors and write the file data to an array as well. read_input_three() is used to parse argument 3's file and returns the number of nodes, and stores the input file data in the four arrays: nodeVisited_h,nodeGate_h, nodeInput_h, and nodeOutput_h. Lastly, argument four is passed to read_input_one_two_four() and returns the number of current level nodes, and stores this data locally. Arguments 5 and 6 represent output files, which are opened with write only. The last line of preparation before execution is allocating memory for the next level nodes, which is the size numTotalNeighbors, found when parsing the second argument file.

We began our clock timing here. A for loop iterates through all the current level nodes, the length of the fourth argument. an active node is taken from currLevelNodes array at the loops index i. A nested loop occurs which iterates from the value in nodePtrs at the index of the node value, until the value at index node+1. This loop iterates through the neighbors of the node found in the outer loop. The neighbor is checked if it has been visited so far. If it has been visited before, no action is taken and the loop continues. Otherwise, it is set to visited and the gateSolver method is called inorder to find the output of the node. Next nextLevelnodes_h at the index numNexLevelNodes_h is set to the value held in neighbor, and numbNextLevelNodes_h is incremented before the loop continues.

After the two nested loop conclude, the clock ends and the execution time is printed to the console. The first output file is filled with all the data in the Node Output array, and the second input file is filled with all the data in the nextLevelNodes_h array.

At the conclusion of the program, all memory assigned for the computations of BFS are freed, then the program terminates.

Below is the execution time of our sequential computations BFS program.

Table 1: Execution time for sequential BFS

Time (ms)
1.820000

2. Global queueing parallelization of BFS

   The Global parallelization scheme begins the same way as our sequential scheme. The inputs are parsed in the same way. Once the inputs are parsed, cudamalloc is used to assign memory for all the input and output arrays. After the memory is allocated, the memory is copied from the host's arrays to the device for parallel computations. After all the memory is allocated and copied over, the Number of blocks and threads per block is assigned, which will be changed in Table 2 to see how those values affect execution time.

   At this point our timer starts and the parallelization portion of the program begins with the assigned number of blocks and threads per block. Each thread executes the method global_queuing_kernel(). The number of blocks, threadsperBlock number of current level nodes and the input and output cuda arrays are passed to each thread. First the thread finds it's id. Then a for loop begins, which iterates from the thread id to the number of current level nodes. each pass of the loop the loop variable i is incremented by the number of threads per block times the number of blocks. The active node of each loop is the value in currLevelNodes_h at the index of i. Here a nested loop occurs that iterates over that node's neighbors and operates on them in the same way done in the sequential program. After the gate_solver() call, a position is found by using AtomicAdd on numNextLevelNodes_h. Then the atomicExch() value is called with the value address of nextLevelNodes at the given position and the active neighbor.

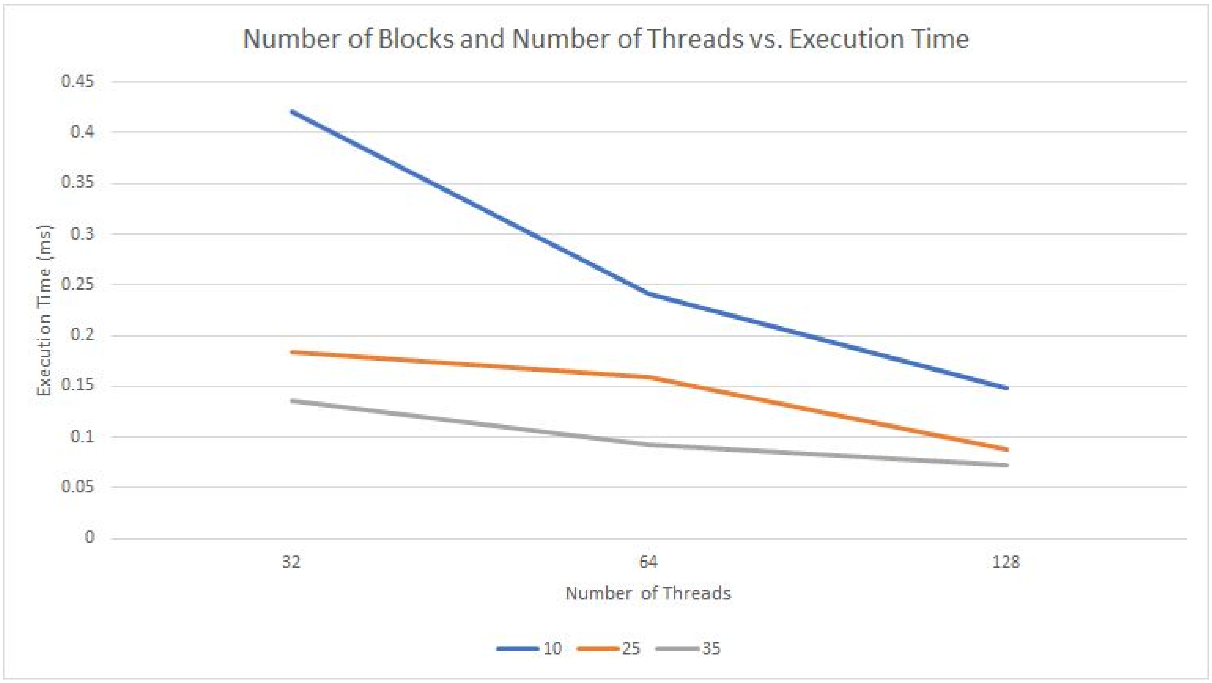
   When all blocks and threads conclude their executions, the arrays holding next level nodes, node outputs, and numNextLevelNodes are copied back from the device to the hosts for writing the output to the output files. The program concludes in the same way as our sequential program, by writing nodeOutputs to the first output file and nextLevelNodes to the second. The execution time is printed to the console and the cuda memory is freed. The program closes the output files then terminates.

   Below in table 2, displays the relationship between the number of blocks and the number of threads and the Execution time of the parallel portion of the execution. Our finding was that, when the number of threads increases on a static number of blocks, the execution time decreases, as shown in figure 1. When the Number of Blocks increased, the execution time decreased more significantly.

Table 2: Execution time for Global queueing BFS

Number of blocks	Number of threads	Execution time (ms)
10	32	0.420992
10	64	0.240448
10	128	0.148672
25	32	0.184064
25	64	0.158400
25	128	0.088352
35	32	0.135200
35	64	0.092064
35	128	0.071744

Figure 1: Number and Blocks and Number of Threads vs. execution on a Global queueing parallelization scheme



3. Block queuing parallelization of BFS

For block queuing, we created three global variables that are used by the GPU: a global counter device variable, that counts the number of nodes in the global queue, a block counter shared variable, that counts the nodes processed by a block, and a current counter shared variable, that counts the nodes that have been added to a block queue.

Within the main method, the execution of block queuing and global queuing is similar, except that in Block queuing we call the kernel method more than once so that each block takes care of the same number of nodes. So we keep track of how many times we have called it with a "round" variable that is incremented, and a "numRounds" constant that is the number of nodes in the level divided by the total number of threads.

In the kernel function, we initialize a queue and an index to keep track of which range of node's neighbors we are processing. We check that the thread ID is within the total number of threads, and we identify the node with its index that was previously declared.

Then, we go over a for loop similar to the global queuing part, where we iterate over the neighbors of the node. For each neighbor, we check if it has been visited. If not, we mark it as visited and calculate its gate output with the current node. Then, we add it to the block queue if it isn't full, and to the global queue if the block queue is full. After we have looped over the neighbors, if we have completed the rounds, one thread per block copies the elements of the block queue into the global queue.

Unfortunately, our implementation doesn't work when using a queue capacity of 64. We think this issue has to do with how we count the rounds and we keep count of the different queue sizes, but unfortunately we were not able to solve the problem.

Below is the execution time for different numbers of blocks, threads per block and queue capacity (table 3). We observed that the execution was a lot longer than with global queuing, which is another indication that something goes wrong with our implementation. Furthermore, as expected, we see that the execution time decreases when the number of threads increases, as well as when the number of blocks increase.

Table 3: Execution time for Block queueing BFS

Number of blocks	Number of threads	Queue capacity	Execution time (ms)
25	32	32	0.251936
25	32	64	-
25	64	32	0.170656
25	64	64	-
35	32	32	0.182784
35	32	64	-
35	64	32	0.127328
35	64	64	-