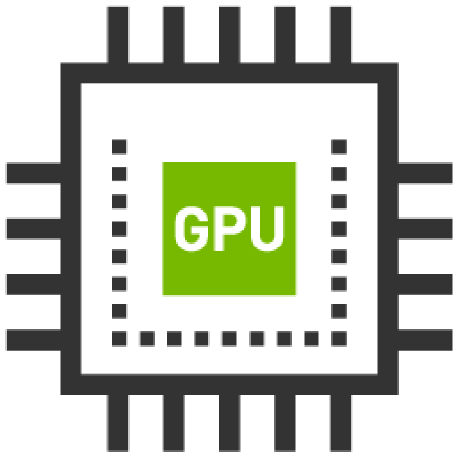
ECSE 420 - Parallel Computing
Lab 3 Report



Group 45

Ahmed Azhar – 260733580

Garrett Kinman – 260763260

Table of Contents

Table of Contents **1**

Architecture **2**

Breadth-First Search **2**

Sequential Implementation 2
Parallel with Global Queuing Implementation 2
Parallel with Block Queuing Implementation 3
Analysis of Experimental Results 4

Architecture

For reference, the results were obtained using a host device on Google Colab that provides a 12GB NVIDIA Tesla K80 GPU, having 2496 CUDA cores. Furthermore, this GPU is also capable of 1024 threads per block.

Breadth-First Search

In this lab, several implementations of a breadth-first simulation of logic gates were created. In this, each logic gate was treated as a node in a graph, and then, during a breadth-first traversal, all logic gate nodes were simulated. The first implementation was a sequential implementation, then it was parallelized, first using a global queue, then using block queues. These implementations are detailed below.

Sequential Implementation

In the sequential implementation, the program loops over all the nodes in the current level, checks the neighbors of each, and adds any unvisited neighbors of those to the queue, updating the simulated output along the way.

```
//BFS Loop
clock_t begin_timer = clock();
// Loop over all nodes in the current level
for (int i = 0; i < numCurrLevelNodes; i++) {
    int node = currLevelNodes[i];

    // Loop over all neighbors of the node
    for (int j = nodePtrs[node]; j < nodePtrs[node+1]; j++) {
        int neighbor = nodeNeighbors[j];

        // 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(nodeInput[neighbor], nodeOutput[node], nodeGate[neighbor]);
            nextLevelNodes[numNextLevelNodes] = neighbor;
            ++numNextLevelNodes;
        }
    }
}
clock_t stop_timer = clock();
```

Below, we have the execution time for the sequential implementation.

Execution Time (ms)
2.187

Parallel with Global Queuing Implementation

In the parallel implementation with global queuing, the outer loop was parallelized, with each thread being assigned a certain number of nodes in the current level to loop through. Because more than one node can share a neighbor, leading to the possibility of a node being accessed by more than one thread simultaneously, atomic operations were used in assessing whether it was visited and for updating the global queue index.

```
__global__ void global_queuing_kernel(int totalThreads, int countNodes, int* nodePtrs, int* currLevelNodes, int* nodeNeighbor
    int nodesPerThread = countNodes / totalThreads;
    int threadIndex = threadIdx.x + (blockDim.x * blockIdx.x);
    int beginIdx = threadIndex * nodesPerThread;
    //Loop over all nodes in the current level
    for (int id = beginIdx; id < countNodes && id < beginIdx + nodesPerThread; id++) {
        int nodeIdx = currLevelNodes[id];
        //Loop over all neighbors of the node
        for (int secondId = nodePtrs[nodeIdx]; secondId < nodePtrs[nodeIdx+1]; secondId++) {
            int neighborIdx = nodeNeighbors[secondId];
            //If the neighbor hasn't been visited yet
            const int visited = atomicExch(&(nodeVisited[neighborIdx]),1);
            if (!visited) {
                nodeOutput[neighborIdx] = gate_solver(nodeGate[neighborIdx], nodeOutput[nodeIdx], nodeInput[neighborIdx]);
                //Add it to the global queue
                const int globalQueueIdx = atomicAdd(&numNextLevelNodes,1);
                globalQueue[globalQueueIdx] = neighborIdx;
            }
        }
        __syncthreads();
    }
}
```

One limitation of this implementation is the lack of nested parallelism. Because there are two nested for loops, and only the outermost one is parallelized, greater performance could likely be achieved if, for every node a thread visits, it can generate threads for each of its neighbors. Currently, it has to iterate through the neighbors sequentially.

Below, we have the execution times for several combinations of block size and number of blocks. We observe a rather consistent execution time of ~15–18 μs, compared to the 2.187 ms of sequential. This represents a speedup of approximately 137 times. This speedup is for all the tested permutations of block size and number of blocks below.

Block Size	**Number of Blocks**	**Execution Time (ms)**
32	10	0.015
32	25	0.018
32	35	0.016
64	10	0.017

64	25	0.016
64	35	0.016
128	10	0.017
128	25	0.016
128	35	0.016

Parallel with Block Queuing Implementation

In the parallel implementation with block queuing, the parallelization scheme is essentially the same as for with global queuing, as described above. The key difference is that, instead of adding unvisited neighboring nodes to a globally scoped queue that is visible throughout the device, these nodes are instead added to a block-scoped queue that is visible only within the same block. Only after the block queue has filled does is it all added to a global queue.

```
__global__ void block_queuing_kernel(int numCurrLevelNodes, int* currLevelNodes, int* nodeNeighbors, int* nodePtrs, int* nodeV
    // initialize shared memory queue
    extern __shared__ int sharedBlockQueue[];
    __shared__ int sharedBlockQueueSize, blockGlobalQueueIdx;

    if (threadIdx.x == 0)
        sharedBlockQueueSize = 0;

    __syncthreads();
    int threadIndex = threadIdx.x + (blockDim.x * blockIdx.x);
    // Loop over all nodes in the current level
    for (int id = threadIndex; id < numCurrLevelNodes; id++) {
        int nodeIdx = currLevelNodes[id];
        //Loop over all neighbors of the node
        for (int nId = nodePtrs[nodeIdx]; nId < nodePtrs[nodeIdx+1]; nId++) {
            int neighborIdx = nodeNeighbors[nId];
            // If the neighbor hasn't been visited yet
            const int visited = atomicExch(&(nodeVisited[neighborIdx]), 1);
            if (!(visited)) {
                const int queueIdx = atomicAdd(&sharedBlockQueueSize, 1);
                // Solve Gate
                nodeOutput[neighborIdx] = gate_solver(nodeGate[neighborIdx], nodeOutput[nodeIdx], nodeInput[neighborIdx]);
                // if not full add to block queue
                if (queueIdx < queueSize){
                  sharedBlockQueue[queueIdx] = neighborIdx;
                }
                else { // else, add to global queue
                    sharedBlockQueueSize = queueSize;
                    const int GlIdx = atomicAdd(&numNextLevelNodes, 1);
                    nextLevelNodesQueue[GlIdx] = neighborIdx;
                }
            }
        }
    }
```

The limitation of no nested parallelism as described above is also present in this block queuing variant.

Below, we have the execution times for several combinations of block size and number and block queue capacity. We observe a significantly slower execution time than for global queuing. Similar to global queuing, however, we observe a consistent execution time for all the tested permutations of block size, number of blocks, and block queue capacity, all within the range of 39.3–39.5 ms.

Block Size	**Number of Blocks**	**Block Queue Capacity**	**Execution Time (ms)**
32	25	32	39.323
32	25	64	39.328
32	35	32	39.336
32	35	64	39.343
64	25	32	39.450
64	25	64	39.456
64	35	32	39.467
64	35	64	39.463

Analysis of Experimental Results

In our global queuing variant of the parallelized algorithm, we observed a large, consistent (with respect to the permutations of tested metaparameters) speedup of about 137 compared to the sequential implementation. Despite no obvious differences in the algorithms (besides the matter of global vs block queuing), block queuing has massively slower execution times than either sequential or global queuing.