CS406 Final Project Report

Kerem Yildirir , Yiğit Aras Tunalı, Cem Alptürk

May 2019

1 Problem Definition

Breadth-first search is the graph traversal algorithm which systematically traverses the graph starting from a provided starting node and visits each node s that can be reached from the starting node. The algorithm does this exploration by visiting first the nodes that are k, $k \in 1..n$ where n can be any positive integer until all nodes are visited, distanced to the root, i.e. the starting Node, and marking them, then visiting distanced nodes afterward. The Nodes that are being processed and residing in the data structure are called the "Frontier" and every $k+1$ distanced nodes will be added to the FIFO, first-in-first-out, data structure one by one for the future steps of traversal. This procedure will continue as k is incremented 1-by-1 until the queue is empty and all of the reachable nodes are visited.

The features of the data sets we are using to test our implementations are as follows;

Features of Given Graphs					
Feature	**coPapersDBLP**	**wiki-topcats**	**europe_osm**	**rmat-er**	**rmat-b**
# of vertices	540486	1791489	50912018	16777216	16777216
# of edges	30491458	28511807	108109320	268435308	268435308
avg outdegree	56.41	15.9151	2.12345	16	16
depth	14	257	13847	8	7

2 Challenges

The challenges of the parallelization of the BFS algorithm are caused by the nature of the algorithm. In essence, the BFS algorithm is a memory-bounded problem, and there are no heavy computations that are being done that can be used to hide behind while using parallelization. Another major challenge for the parallelization is the random access nature of the BFS, the workload distribution is not known until the run time, and this causes workload imbalances across threads while the algorithm is being run in parallel. The features of the graphs such as, dense or sparse graphs, the depth of the graph, possible frontier sizes and all the variability between graphs make it harder to implement algorithms that work in general case, so specialized implementation tricks are necessary in most cases. All these problems combined make the BFS algorithm challenging to parallelize.

3 Naive Sequential Algorithm

Breadth First Search is a graph traversal strategy for visiting all the vertices of a graph. The serial algorithm for the problems is as follows;

Algorithm 1 Serial BFS Algorithm

```
1:  for v ∈ V do
2:      v.dist = ∞
3:  Q.enqueue(v_0)
4:  v_0.dist ← 0
5:  while Q is not empty do
6:      u ← Q.dequeue()
7:      for all w ∈ adj(u) do
8:          if w.dist == ∞ then
9:              w.dist = u.dist + 1
10:             Q.enqueue(w)
```

The parallelization of the algorithm brings about some challenges that need to be addressed. Since the nature of the algorithm is more memory bounded and not dense as in the computations required, it is hard to mask this memory bandwidth with the parallelization of computation. The memory access pattern and workload distribution is also data dependent and can be considered random as well, which adds more challenges to the efficient implementation of the algorithm[1]. There are many techniques followed in the literature such as; approaches that focus on containers, strategies that focus on vertex parallelization and methods that implement the parallelization without synchronization steps by increasing the overall complexity [2]. We acquired a better sequential performance in our final version on CPUs, hence we compare our speedup and efficiency values according to our best version, since it is the best sequential implementation we have for this problem.

Our Best Sequential Implementation vs Given Benchmarks		
Data set	**Given Benchmark**	**Our Implementation**
rmat-er	3.94	1.05
rmat-b	2.212092	-
europe-osm	1.306992	1.59
coPapersDBLP	0.056	0.036
wiki-topcats	0.174	0.072

Table 1: Our best sequential implementation comparison with the sequential benchmark for the given graphs

4 Steps of Parallelization

4.1 CPU Implementations

The specs of the CPU's used in the Nebula are as follows (there are 32 cores of the same type);

Model name: Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz
Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
CPU(s): 32
Thread(s) per core: 2
Core(s) per socket: 8
Socket(s): 2
NUMA node(s): 2
Vendor ID: GenuineIntel
CPU MHz: 1200.117, **CPU max MHz:** 3000.0000, **CPU min MHz:** 1200.0000
L1d cache: 32K, **L1i cache**: 32K, **L2 cache:** 256K, **L3 cache:** 20480K

4.1.1 Thread-Safe Queue Implementation

The first approach we have taken was to implement a global, thread-safe queue. We used the BOOST library's thread-safe queue implementation for this approach, but unfortunately, it performed very poorly. Due to the queue being a hot spot for congestion, there wasn't a decent speed-up. Thus we had to abandon this implementation pretty quickly.

4.1.2 Array Based Implementation

The second approach we have taken was fundamentally similar to the global thread-safe queue approach. But instead of one queue, we decided to assign a 2D matrix of size $t * N$, where t is the number of threads, and N is the size of the vertex array. We assumed that we could assign each thread its queue and use this approach to parallelize the work efficiently. This approach quickly failed us, as we realized that the workload distribution was arbitrary and a thread ending up with 0 children would stay idle unless a synchronization step were included. This was true for workload balancing as well, but implementing a synchronization step would mean introducing a really time consuming critical region, thus slowing this already slow approach. We have moved away from this approach pretty quickly after this realization.

4.1.3 Search-based Implementation

Since in the last step, the central part that reduced our performance was synchronization and locks, with our next application, we tried to avoid using them. The reason we chose this strategy was to get rid of memory allocations at each level, as well as using a single array for the whole implementation, reducing the cache-miss rate. For this approach, we used a length N array, initially all set to ∞. Using the CRS, compressed row storage, format to store the graph, we traversed the array split into chunks by $schedule(static)$ and check all the elements. If an element has a distance value of the level, we are currently at, the threads mark their children by $dist + 1$, and when the traversal of the array is finished, the next level starts. By following this approach step-by-step, we increase the number of steps necessary in each loop but avoid using a container which would require synchronization. The results acquired by this approach was better than the previous but still as the number of threads increased the speed-up we received increased very slowly. The reason for this slow improvement seems to be the random memory access nature of the algorithm coupled with O(N/p) search operation done at each level(p is the number of threads). It also performs abysmally when N gets very large, and frontier size is minimal.

Algorithm 2 Search Based BFS

```
1:  for v ∈ V do
2:      distances[v] = ∞          ▷ Keep an array of N elements where each index i is mapped to ith vertex
3:  Select root
4:  distances[root] = 0
5:  for all v adjacent to root do in parallel
6:      distances[v] = 1
7:  end
8:  dist = 0
9:  while changed is true do
10:     changed = false
11:     for i = 0 → N in parallel
12:         if distances[i] == dist then
13:             for  v ∈ adj(vertexi) do
14:                 if Vertex v is not visited then
15:                     distances[v] = dist + 1
16:                     changed = true
17:     end
18:     dist++
```

The results for the Search Based BFS are as follows, note that the algorithm performed very poorly in europe_osm and failed to execute in a reasonable time since the graph is very deep (There was also a small problem with counting MTEPS in this approach so we couldn't for search based method).

CPU Search Based method								
	Execution Time				Speedup			
Threads	coPapers	wiki-topcats	europe_osm	rmat-er	coPapers	wiki-topcats	europe_osm	rmat-er
t=2	0.06573	0.36725	-	2.447	0.55	0.1986	-	0.430
t=4	0.05274	0.24303	-	1.464	0.069	0.3001	-	0.719
t=8	0.04103	0.16274	-	1.001	0.087	0.4482	-	1.05
t=16	0.03261	0.1323	-	0.84	1.125	0.5513	-	1.25

CPU Search Based method								
	Efficiency				MTEPS			
Threads	coPapers	wiki-topcats	europe_osm	rmat-er	coPapers	wiki-topcats	europe_osm	rmat-er
t=2	0.275	0.0993	-	0.215				
t=4	0.017	0.075	-	0.179				
t=8	0.0108	0.056	-	0.131				
t=16	0.0703	0.0344	-	0.078				

Table 2: CPU Search Based Method implementation results.

4.1.4 Top-Down Implementation

The top-down approach is the traditional implementation for BFS, as mentioned in [3], it is very similar to the naive approach. However, in parallel implementation, each vertex in the frontier queue is processed in parallel. This was done by using local queues for each thread, then merging them at the end of each level, constructing the next frontier. We were also able to avoid the usage of any critical region by using prefix sums to compute the target index for each thread while writing to the next frontier in parallel. This approach is useful when we have a moderate amount of nodes at each level, but it becomes slows down when the frontier size becomes large. This is due to the fact that in top-down implementation, we have to check each neighbor of every node in the frontier to check if they are visited or not and frontier size grows. We are doing a lot of unnecessary edge checks for already visited vertices. The results for only the top-down approach can be seen in the following table.

CPU Top-down Method								
	Execution Time				Speedup			
Threads	coPapers	wiki-topcats	europe_osm	rmat-er	coPapers	wiki-topcats	europe_osm	rmat-er
t=2	0.04248	0.1277	1.006	2.2381	0.085	0.571	1.159	0.47
t=4	0.0269	0.0732	0.7029	1.2028	1.384	0.996	2.266	0.875
t=8	0.01959	0.0445	0.5850	0.7127	1.894	1.637	2.719	1.47
t=16	0.01499	0.0403	1.039	0.6469	2.571	1.808	1.545	1.64

CPU Top-down Method								
	Efficiency				MTEPS			
Threads	coPapers	wiki-topcats	europe_osm	rmat-er	coPapers	wiki-topcats	europe_osm	rmat-er
t=2	0.0425	0.285	0.579	0.235	717.78	223.27	107.46	119.93
t=4	0.346	0.249	0.566	0.218	1133.58	389.52	153.80	223.17
t=8	0.236	0.204	0.339	0.183	1556.5	640.76	184.81	376.64
t=16	0.16	0.113	0.096	0.102	2035.16	707.59	104.06	414.95

Table 3: CPU Top-down Method implementation results.

4.1.5 Bottom-Up Implementation

Bottom-up BFS is an alternative to the top-down approach, proposed by [3]. In this approach, we visit the neighbors of each unvisited vertex and check if any of their neighbors are in the frontier if so, we stop the traversal for that vertex and add it to the frontier since it found a parent to itself. The beauty of this approach comes from its simplicity and its effectiveness when frontier size is large. However, only using this approach is not beneficial, since when frontier size is small, we have to traverse a lot of unnecessary vertices while looking for membership in the frontier. The results for only the bottom-up approach can be seen in the following table.

CPU Bottom-up Method								
	Execution Time				Speedup			
Threads	coPapers	wiki-topcats	europe_osm	rmat-er	coPapers	wiki-topcats	europe_osm	rmat-er
t=2	0.1338	0.1879	-	4.0806	0.276	0.3881	-	0.257
t=4	0.0875	0.1089	-	2.0118	0.36	0.669	-	0.522
t=8	0.0761	0.0620	-	1.2303	0.473	1.175	-	0.853
t=16	0.0444	0.0407	-	1.1296	0.081	1.788	-	0.937

CPU Bottom-up Method								
	Efficiency				MTEPS			
Threads	coPapers	wiki-topcats	europe_osm	rmat-er	coPapers	wiki-topcats	europe_osm	rmat-er
t=2	0.138	0.194	-	0.1285	994.44	642.19	-	365.57
t=4	0.09	0.167	-	0.13	1519.3	883.93	-	741.51
t=8	0.059	0.146	-	0.106	1660.2	1344.4	-	1179.9
t=16	0.005	0.111	-	0.058	2085.3	1832.1	-	1034.5

Table 4: CPU Bottom-up Method implementation results.

4.1.6 Direction Optimizing BFS (Hybrid)

In above two sections, we described 2 different approaches for BFS. In this final implementation, we are combining [3]'s bottom-up approach and the traditional top-down approach. With this version, BFS algorithm works with top-down approach when frontier size is small, and switches to bottom-up approach when frontier size gets big enough. [3] used special α and β parameters for finding the best switching point between top-down and bottom-up approach, but we followed a simpler method. Given the statistics of our graphs, we decided that it is more beneficial to use top-down approach when number of vertices in frontier is less than 5% of total number of vertices, and used it as the threshold for deciding when to switch.

With this implementation, we used the best approach for the current frontier at any time, and acquired the best results on CPU with 16 threads with below timing and speedup values. We also acquired our best sequential results using this method, hence all of our speedup calculations are based on this method.

CPU Hybrid Approach								
	Execution Time				Speedup			
Threads	**coPapers**	**wiki-topcats**	**europe_osm**	**rmat-er**	**coPapers**	**wiki-topcats**	**europe_osm**	**rmat-er**
t=2	0.0233	0.0532	1.0027	0.6375	1.565	1.370	1.591	1.648
t=4	0.0167	0.036	0.6931	0.4474	2.25	2.023	2.29	2.348
t=8	0.0108	0.0238	0.5831	0.2458	3.6	3.054	2.72	4.285
t=16	0.0082	0.0225	1.029	0.2151	4.5	3.229	1.558	5.47

CPU Hybrid Approach								
	Efficency				MTEPS			
Threads	**coPapers**	**wiki-topcats**	**europe_osm**	**rmat-er**	**coPapers**	**wiki-topcats**	**europe_osm**	**rmat-er**
t=2	0.782	0.685	0.795	0.824	665.59	110.98	107.81	254.17
t=4	0.562	0.5057	0.572	0.587	847.11	128.72	155.98	362.17
t=8	0.45	0.381	0.34	0.535	1243.98	116.63	185.42	659.22
t=16	0.281	0.201	0.0097	0.341	476.36	84.96	105.07	561.95

Table 5: CPU Hybrid Method implementation results.

4.2 GPU Implementations

We used the following GPUs:(We used N/1024 blocks and 1024 thread on each block.)

GTX TITAN X Engine Specs:
3072 CUDA Cores
1000 Base Clock (MHz)
1075 Boost Clock (MHz)
192 Texture Fill Rate (GigaTexels/sec)
GTX TITAN X Memory Specs:
7.0 Gbps Memory Clock
12 GB Standard Memory Config
GDDR5 Memory Interface
384-bit Memory Interface Width
336.5 Memory Bandwidth (GB/sec)

GTX 980 Engine Specs:
2048 CUDA Cores
1126 Base Clock (MHz)
1216 Boost Clock (MHz)
144 Texture Fill Rate (GigaTexels/sec)
GTX 980 Memory Specs:
7.0 Gbps Memory Clock
4 GB Standard Memory Config
GDDR5 Memory Interface
256-bit Memory Interface Width
224 Memory Bandwidth (GB/sec)

(Note that the charts in below subsections are the results from TITAN X GPU device)

4.2.1 Search Based Implementation

Since we have enough threads for assigning each thread to a vertex in the graph, we implemented the search based approach we tried in CPU since we are not using any frontier queue, we are not sending a large amount of data back and forth between GPU and CPU during the iterations. The only thing we need to check after each iteration is the size of the "frontier" in the next iteration(note that there is no actual frontier queue, this is implemented as a stopping criterion when the frontier size is 0, that means BFS is completed. This is calculated by reducing the number of vertices whose distance value has been changed in the current iteration).

Results of this implementation are as follows;

Algorithm 3 Search Based Approach Kernel

```
ind ← threadIdx.x + blockIdx.x * blockDim.x
v_ind ∈ V
if path[v_ind] == dist then
    for all w ∈ adj(v_ind) do
        if w is unvisited then
            path[w] = dist + 1
```

CUDA Search Based Method					
Graphs	**coPapersDBLP**	**wiki-topcats**	**europe_osm**	**rmat-b**	**rmat-er**
Execution Time	0.006545	0.035451	13.86	-	0.197512
Speedup	5.53	2.0575	0.11	-	5.329
MTEPS	4658.73	814.48	7.8	-	1359.08

Table 6: CUDA Search Based Method implementation results

4.2.2 Top-Down Implementation

This implementation is similar to the Search Based, we have explained in the previous section. The main difference is the threads assigned to the kernel are dynamically calculated. Also, there is a frontier here to keep track of the vertices that need to be processed. Since cores are weaker than CPU, this approach does not provide significant speed-up when nodes have many neighbors. The performance of this approach is also slower than search based except when the graph diameter is high, and the number of vertices is huge (europe-osm).

CUDA Top-down Method					
Graphs	**coPapersDBLP**	**wiki-topcats**	**europe_osm**	**rmat-b**	**rmat-er**
Execution Time	0.00982525	0.0304139	0.336288	-	0.279474
Speedup	3.673	2.3983	4.81	-	3.763
MTEPS	5509.56	948.61	335.30	-	961.58

Table 7: CUDA Top-down Method implementation results.

4.2.3 Search-Based Bottom Up Implementation

This approach is a combination of search-based and bottom-up CPU implementations. Just as in the CPU, this approach prevents unnecessary edge checks and works well when the frontier size is big. The main issues with this approach are the imbalanced workload distribution between threads, also the idling of threads after a while. Performance issues arise when the frontier size is small thus this approach doesn't work well alone and needs to be coupled with top-down approach (The run-time explodes on europe_osm graph since the diameter of the graph is, and this approach doesn't work well in that case, so the results for that graph aren't included).

Algorithm 4 Bottom-up Approach Kernel

```
ind ← threadIdx.x + blockIdx.x * blockDim.x
v_ind ∈ V
if v_ind is unvisited then
    for all w ∈ adj(v_ind) do
        if path[w] == dist then
            path[v_ind] = dist + 1
```

Results for this approach are as follows;

CUDA Bottom-up Method					
Graphs	**coPapersDBLP**	**wiki-topcats**	**europe_osm**	**rmat-b**	**rmat-er**
Execution Time	0.0133573	0.212984	38.2	-	0.6719
Speedup	2.769	0.3424	-	-	1.567
MTEPS	7678.82	464.93	16.51	-	1820.72

Table 8: CUDA Bottom-up Method implementation results.

4.2.4 Hybrid Implementation

This implementation is the one that worked well for most of the graphs for us, apart from the europe_osm graph. It is the combination of Top-Down and Bottom-Up approaches discussed in the previous sections. The switching decision is made with parameters that are received from the current level, just as in the CPU implementation. The difference from the CPU implementation is that we didn't need to use a frontier and just used the distance array to keep track of the work required to be done on the next level. The results are as follows;

CUDA Hybrid Method					
Graphs	**coPapersDBLP**	**wiki-topcats**	**europe_osm**	**rmat-b**	**rmat-er**
Execution Time	0.00366	0.0223031	10.319	-	0.027103
Speedup	9.93	3.27048	0.154	-	38.88
MTEPS	2157.16	382.31	10.47	-	1046.86

Table 9: CUDA Hybrid Method implementation results.

5 Final Results and Comparisons

To sum up, we observed our best results with GPU on all graphs. We were able to exploit the massive amount of threads by distributing each vertex of the graph to a single thread, then looking for a parent or a frontier node. Checking membership does not require any kind of search operations, due to every thread checking a single vertex in search based approaches (top-down and bottom-up). However, this approach suffered when number of vertices becomes too large. In such case, traditional top-down approach performs best. In CPU, hybrid approach performed best due to switching the best strategy in each level. Following tables illustrate a summary of all our methods for solving this problem, and a comparison between our best CPU and GPU implementations.

CPU Implementations									
		Execution Time				**Speedup**			
Methods	**Threads**	**coPapers**	**wiki-topcats**	**europe_osm**	**rmat-er**	**coPapers**	**wiki-topcats**	**europe_osm**	**rmat-er**
Search Based	**t=2**	0.0657	0.367	-	2.447	0.55	0.198	-	0.43
	t=4	0.0527	0.243	-	1.464	0.069	0.3001	-	0.719
	t=8	0.041	0.162	-	1.001	0.0087	0.448	-	1.05
	t=16	0.0326	0.132	-	0.84	1.125	0.551	-	1.25
Top Down	**t=2**	0.0424	0.1277	1.006	2.238	0.085	0.571	1.591	0.47
	t=4	0.0269	0.0732	0.7029	1.202	1.384	0.996	2.266	0.875
	t=8	0.0195	0.0445	0.585	0.712	1.894	1.637	2.719	1.47
	t=16	0.0149	0.0403	1.039	0.646	2.571	1.808	1.545	1.64
Bottom Up	**t=2**	0.1338	0.187	-	4.08	0.276	0.3881	-	0.257
	t=4	0.087	0.108	-	2.011	0.36	0.669	-	0.522
	t=8	0.076	0.062	-	1.23	0.473	1.175	-	0.853
	t=16	0.044	0.04	-	1.129	0.081	1.788	-	0.093
Hybrid	**t=2**	0.023	0.053	1.002	0.637	1.565	1.370	1.591	1.648
	t=4	0.0167	0.036	0.693	0.447	2.25	2.023	2.29	2.348
	t=8	0.0108	0.0238	0.583	0.245	3.6	3.054	2.72	4.285
	t=16	0.0082	0.0225	1.029	0.215	4.5	3.229	1.558	5.47

Table 10: CPU implementation results comparison (best ones marked green).

CUDA Implementations								
	Execution Time				**Speedup**			
Method	**coPapers**	**wiki-topcats**	**europe_osm**	**rmat-er**	**CoPapers**	**wiki-topcats**	**europe_osm**	**rmat_er**
Search b.	0.006545	0.035451	13.86723	0.197512	5.53	2.0575	0.11	5.329
Bottom-up	0.0133573	0.212984	-	0.671929	2.769	0.3424	-	1.567
Hybrid	0.00366371	0.0223031	10.319584	0.027103	9.93	3.27048	0.154	38.88
Top-down	0.00982525	0.0304139	0.336288	0.279474	3.673	2.3983	4.81	3.763

Figure 1: CUDA implementation results comparison (best ones marked green).

6 How To Run the Code

There is a Makefile, for CPU implementation please use command "make hybrid_cpu". For GPU implementation like-wise use command "make hybrid_cuda". Also since EUROPE graph is really deep, we needed another approach for it as explained before, so to be able to run the GPU code there is another make file option; "make ercu". Please run the graphs with similarities to europe graph with ercu implementation.

Appendices

```
void top_down_step(int *&frontier,int &load,int *&next,bool &check,int dist)
{
  check = false;
  int size = 0;
#pragma omp parallel //proc_bind(spread)
  {
    int tid = omp_get_thread_num();
    int pos=  0;//indices[tid] = 0;
#pragma omp for schedule(guided)
  for(int i=0;i<load;i++){
      int elem  = frontier[i];
      int start = row[elem];
      int end = row[elem+1];
      for(int j=start;j<end;j++){}
     int ind = col[j];
     if(path[ind] == -1){
         path[ind] = dist;
         local_queues[tid][pos++] = ind;
       }
       }
    }
  indices[tid] = pos;
  int sz = pos;
  #pragma omp critical
   {
     for(int j=0;j<sz;j++){
        next[size] = local_queues[tid][j];
        size++;
      }
   }
 }
  load = size;
  check = load;
  int *temp = frontier;
  frontier = next;
  next = temp;
}
```

Listing 1: Top down step on the CPU implementation

```
__global__ void top_down_step(int *load,int *path,int dist,int *row,int *col)
{
  unsigned int ind = threadIdx.x + blockIdx.x * blockDim.x;

    if(path[ind] == dist){
      int start = row[ind];
      int end = row[ind+1];
      for(int I = start; I < end; I++){
       int i = col[I];
       if(path[i] == -1){
          path[i] = dist+1;
          atomicAdd(load, 1);
       }
     }
    }
}
```

Listing 2: Search-based top down approach on CUDA

```
void bottom_up_step(int *&frontier,int &load,int *&next,bool &check,int dist,boost::dynamic_bitset<>  &myset)
{
  check = 1;
  int size = 0;
#pragma omp parallel //proc_bind(spread)
  {
    int tid = omp_get_thread_num();
    int pos = 0;
#pragma omp for schedule(guided)
  for(int i=0;i<N;i++){
      int elem  = i;
      if(path[elem] == -1){
     int start = row[elem];
     int end = row [elem+1];
     int par = -1;
     for(int i=start;i<end;i++){
         int neigh = col[i];
         if(myset[neigh]){
        par =  neigh;
        break;
       }
    }
     if(par != -1){
         path[elem] = dist;
         local_queues[tid][pos++] = elem;
       }
   }
    }
  indices[tid] = pos;
  }
  double s = omp_get_wtime();
  int *prefix_sum = new int[numThreads];
  prefix_sum[0] = indices[0];
  for(int id=1;id<numThreads;id++)
    prefix_sum[id] = indices[id]+prefix_sum[id-1];
#pragma omp parallel
  {
    int tid = omp_get_thread_num();
    int sz = indices[tid];
    int ps = prefix_sum[tid-1];
    for(int j=0;j<sz;j++)
      {
   next[ps+j] = local_queues[tid][j];
      }
  }
  load = prefix_sum[numThreads-1];
  check = load;
  int *temp = frontier;
  frontier = next;
  next = temp;
 }
```

Listing 3: Bottom-up step in CPU implementation

```
__global__ void bottom_up_step(int *load,int *path,int dist,int *row,int *col)
{
  unsigned int ind = threadIdx.x + blockIdx.x * blockDim.x;
  if(path[ind] == -1){
      int start = row[ind];
      int end = row[ind+1];
      for(int I = start; I < end; I++){
       int i = col[I];
       if(path[i] == dist){
         path[ind] = dist+1;
         atomicAdd(load, 1);
         break;
          }

       }
    }
}
```

Listing 4: Search based bottomup step on CUDA

References

[1] C. E. Leiserson and T. B. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)," in *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, (New York, NY, USA), pp. 303–314, ACM, 2010.

[2] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core cpu and gpu," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, pp. 78–88, Oct 2011.

[3] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, (Los Alamitos, CA, USA), pp. 12:1–12:10, IEEE Computer Society Press, 2012.