CS406 Project - Parallelization of BFS

Baris Sevilmis, Emre Yasin Sivri, Gizem Aydin

May 2019

Abstract

Breadth First Search stands out as a vital Graph Traversal Algorithm. It is a relatively simple algorithm to implement, though the efficiency of the algorithm is highly dependant on the implementation. Therefore, in this project, we aim to increase the efficiency and the speed of the BFS Traversal using various algorithmic approaches such as Top-Down and Bottom-Up BFS[1] with well-done implementations. Upon these unique algorithmic approaches, parallelization of these approaches using both OpenMP and CUDA are performed to increase the throughput of BFS.

1 Introduction

In this project, Breadth First Search (BFS) is parallelized using a multicore CPU and Nvidia GPU. Throughout the project, different approaches were tried and tested on large graphs for both CPU and GPU Parallelizations such as Top-Down BFS and Bottom-Up BFS. These methods with various implementation techniques yielded different results both on CPU and GPU. Therefore the final implementation of parallelized BFS combines the beneficial traits of all of the approaches. Details of these approaches and implementations are discussed in further sections of this report. Nevertheless, graph data for the project and data structures that are used to implement these approaches are provided as well.
In Section 2, Data Preprocessing and utilization of Compressed Row Storage format are being explained. Section 3 provides a brief overview of the BFS algorithm with Top-Down and Bottom-Up approaches. Section 4 ensures Multicore CPU parallelization techniques of BFS and corresponding details. Section 5 continues by giving necessary GPU implementation details about Layer-based and Frontier-based approaches. Following Section 6, provides results of CPU and GPU implementation with execution time, speedup and efficiency graphs, and additionally comparison of CPU and GPU results are ensured as well.

2 Data Preprocessing

As the scope of the project includes improving performance of Breadth First Search (BFS) on large scale graphs, the representation of these graphs are crucial.

2.1 Adjacency List

The basic approach of storing a graph is the Adjacency List instead of Adjacency Matrix. In adjacency matrix, there is a row and column for each vertex. Considering the fact that the graph is unweighted; if there exists an edge from vertex i to j the corresponding entry in adjacency matrix M, $M[i][j]$ is set to 1. All the other entries are 0. On the other hand, in adjacency list, we append vertex j to the list of vertex i. As a consequence, storing nonzero elements with respect to their vertex ids instead of storing everything saves a lot of memory.

While implementing the adjacency list in C++, the data structure *vector of vector of integers* is used. There is a *vector*, that contains *vector of integers*, for each vertex. If there is an edge from vertex i to j, j is added to the *vector* of i.

2.2 Compressed Row Storage

For sparse matrices, usually the number of edges are *O(n)*, where n denotes the number of vertices. Compressed Row Storage (CRS) of a graph uses two arrays. First array (C in Figure 1) holds all the contents of the adjacency list added one by one. Second array (R in Figure 1) corresponds to the index of C, where the vertices that has an edge from that vertex start [3]. If $R[j] = k$ and $R[j+1] = k+5$, it means that there is an edge from vertex j to all vertices stored from $C[k]$ to $C[k+4]$.

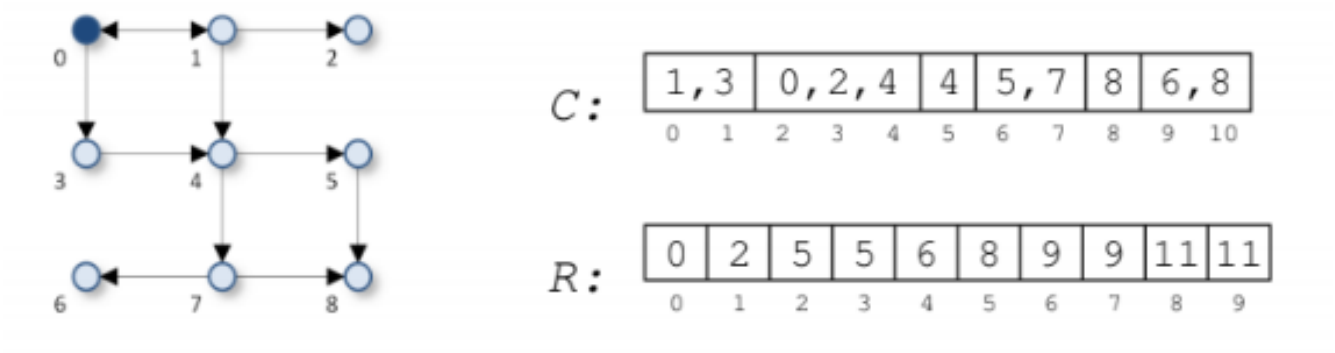

Figure 1: CRS [3]

3 BFS Algorithm

3.1 General BFS Approaches

The underlying BFS algorithms for both parallelization methods are the same. There are three main BFS approaches used in the project; the top-down approach, the bottom-up approach, and the hybrid approach which combines the latter two approaches into a unified approach.

3.1.1 Top Down Approach

The top-down approach is the most well known naïve approach to BFS. It works using a "frontier" method where nodes in the level are placed into a frontier. By iterating over these nodes, the next frontier is found. When iterating over a node, the neighbors of the node's are checked to see if they have been visited. If not, their distance (parent) is updated, and they are placed into the next frontier.

Algorithm 1 BFS Top-Down Traversal in [1]

```
1: procedure TOP-DOWN(vertices, frontier, next, parents)
2:     for v ∈ frontier do
3:         for n ∈ neighbors[v] do
4:             if parents[n] = −1 then
5:                 parents[n] ← v
6:                 next ← next ∪ v
```

3.1.2 Bottom Up Approach

As a consequence, in addition to the Top-Down approach Bottom-Up approach was also proposed in [1,2], in which traversal is in reverse direction. In this approach, instead of each vertex becoming the parent of its neighbors, every unvisited vertex tries to find its parents among its neighbors. As the child writes only to itself in this process, atomic operation requirements are minimized, however Bottom-Up traversal may result in more work in case of small frontiers. Besides, if the graph being used is a directed graph, Bottom-up approach requires the inverse graph. Since the usage of the inverse graph doubles the memory requirements of storing the graph, this should be taken into consideration as well. Algorithm 2 demonstrates Bottom-Up approach on BFS.

Algorithm 2 BFS Bottom-Up Traversal in [1]

```
1: procedure BOTTOM-UP(vertices, frontier, next, parents)
2:     for v ∈ vertices do
3:         if parents[v] = −1 then
4:             for n ∈ neighbors[v] do
5:                 if n ∈ frontier then
6:                     parents[v] ← n
7:                     next ← next ∪ n
8:                     break
```

3.1.3 Hybrid Approach

Considering both advantages and disadvantages of Algorithm 1 & 2, in [1] final decision is to use a hybrid heuristic. To be more specific, heuristic switches between Top-Down and Bottom-Up approaches based on number of vertices in frontier. The benefit of hybrid approach is described in [1] as:

The pairing of the top-down approach with the bottom-up approach is complementary, since when the frontier is its largest, the bottom-up approach will be at its best whereas the top-down approach will be at its worst, and vice versa.

There are many proposed methods on when to change the approach. We used the method proposed in [2], where a frontier which has fewer than (*total number of vertices*/32) vertices is considered a sparse frontier. In that case, we call the top-down-function. However, if it is a dense frontier which means that *frontier size* < (*number of vertices*/32), then the bottom-up approach is called.

4 Multicore-CPU Parallelization With OMP

4.1 General Implementation

The general implementation of OMP BFS is similar to the pseudo-codes given in *BFS Algorithms* section. The main differences are the usage of the results array instead of parents array and usage of prefix sum. The parents array is replaced by results array since the result array holds the distances while also functioning as the parents array. The outer for loop, where traversal of vertices in the frontier is done, is parallelized.

4.2 Improvements on Naive Approach

4.2.1 Prefix Sum

In the BFS algorithm, frontiers are used, as explained previously. However, when executing the Top-Down or the Bottom-up stage, we need to create a new frontier with the new visited child node. However, when all threads run at the same time, this creates race conflicts. Initially, this problem was solved with the usage of *critical* when writing to the new frontier. But this slowed down the execution significantly. Therefore the prefix method was implemented. Each thread writes the vertices that should be put into the frontier into their own frontier arrays called *temp*. When all threads finish the traversal, all threads waiting for each other are ensured with OMP barrier, they calculate where they should write their temp values into the shared frontier. This calculation is done with the usage of thread id and a counter called *tempCtr*, which counts the number of vertices in the threads own *temp* array. Since the calculation of where to write in the frontier array, and the writing of the threads *temp* values into the frontier array are done in parallel; the prefix method is very efficient and useful.

4.2.2 Bit-set for Bottom Up

In the Bottom-up method, for each unvisited neighbor, a check is needed to see whether the neighbor is the frontier array. However, if done inefficiently, this check increases the complexity and runtime significantly. To overcome these problems, the neighbor frontier check had three significant changes on that part:

1. Iterate over all frontier array for each neighbor. This was the first method that was used and was the easiest, yet worst approach in terms of both complexity and runtime.

2. Keep a frontier_check dynamic integer array, which is just read only to check if a neighbor is in the frontier in *O(1)* time. In previous approach, complexity was: *O(num_of_vertices*(max(neighbor count))*front_size).*

3. Changing the frontier_check array from dynamic INT array to bitset array. This way, instead of allocating 8 bit for just one vertex, we allocate 1 bit. Since we only have the values 0 and 1 to signify if it is in frontier or not, bitset structure is very beneficial in this case.

5 GPU Parallelization

CUDA framework is utilized for GPU implementation of BFS algorithm. There exist two main implementation approaches, namely Layer and Frontier approaches, for which details will be explained in following subsections. For both approaches, both Top-Down approach and Hybrid-Approach are distinctly used and tested as in OpenMP parallelization.

5.1 Layer-based Approach

Layer Implementation Approach is convenient for graph structures with relatively few amount of layers. As layer amount increases, to be more specific more distant the initial vertex from other vertices, efficiency of this approach decreases.
For this approach, a layer variable v is initialized as 0, denoting BFS start level. Threads are spread over the *distance* array, which contains distance of each vertex from initial vertex. If there are not enough threads, then threads need to process multiple elements where new elements to be processed are $Current\_Element + Total\_Thread\_Amount$. At every iteration of the main BFS loop following are done for Top-Down and Hybrid approaches:

1. **Top-Down Approach**: Top-Down kernel is launched with parameters
$<<< Ceil(Vertex\_Amount/Thread\_Amount), Thread\_Amount >>>$. Every thread picks vertex corresponding to their global thread id and checks whether $thread\_id == v$. If TRUE, then process each of its neighbors and mark their distances as $v + 1$.Else, corresponding thread remains idle.

2. **Hybrid Approach**: As in OpenMP hybrid parallel implementation, CUDA hybrid implementation follows a similar approach. If *frontier size* < (*number of vertices*/32), then Top-Down kernel is launched. Otherwise, Bottom-Up kernel is launched. Each thread checking an unprocessed element with $distance == -1$ and one of its parents have $thread\_id == v$, then distance of the vertex is updated as $v + 1$.

5.2 Frontier Approach

Frontier Approach is very similar to OpenMP parallel approach, namely keeping frontiers instead of checking vertices in a specific layer. This approach is utilized with a large amount of layers since kernel launch is minimized.

For this approach, there exist $frontier$ and $next\_frontier$. Threads are not spread across the whole result vector but instead spread across the current frontier. Therefore, fewer threads are used at every kernel launch compared to Level-based approach. At every iteration of the main BFS loop following are done for Top-Down and Hybrid approaches:

1. Top-Down Approach: Top-Down kernel is launched with parameters
$<<< Ceil(Frontier\_Size/Thread\_Amount), Thread\_Amount >>>$. Threads are spread across the $frontier$, in which every thread picks the corresponding vertex from the frontier and processes its every neighbor. Every neighbors distance is updated as $current\_distance + 1$. In addition to distance update, a new vertex is placed into the new frontier to process at the next iteration of the main BFS loop. To prevent insertion conflicts, *atomicAdd* is used for updating $next\_frontier$ indices and $next\_frontier$ size.

2. Hybrid Approach: As in OpenMP hybrid parallel implementation, CUDA frontier based hybrid parallel implementation follows a similar approach. If *frontier size* < (*number of vertices*/32), then Top-Down kernel is launched. Otherwise, Bottom-Up kernel is launched as
$<<< Ceil(Vertex\_Amount/Thread\_Amount), Thread\_Amount >>>$ same as in Layer-based approach because every vertex needs to be checked. If the vertex is not processed and its parent is already processed, then its distance is updated as $parent\_distance + 1$. Additionally, this newly processed vertex is added to $new\_frontier$ using again *atomicAdd* operation.

6 Results

6.1 Multicore-CPU Results

Note that the corresponding serial execution (Top-Down to Top-Down and Hybrid to Hybrid) was taken into account while calculating speedup and efficiency. Figure 2 depicts execution times of Top-Down and Hybrid BFS algorithms on 5 different graphs stuctures. On the other hand, Figure 3 demonstrates Speed-Up graphs of Top-Down and Hybrid BFS algorithms on the same graph structures. In addition, Table 1 provides specific execution time and speedup values of Top-Down and Hybrid algorithms. Before interpreting results, it is crucial to provide CPU platforms architecture information, therefore below CPU architecture information are depicted:

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
CPU MHz: 1200.117
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 20480K

Multicore-CPU OpenMP Implementation							
Graph	Threads	TopDown (s)	TD (Speedup)	TD (Efficiency)	Hybrid (s)	H (Speedup)	H(Efficiency)
coPapers	1	0,0564	1	1	0,0411	1	1
	2	0,06	0,94	0,47	0,028	1,46786	0,73393
	4	0,0365	1,5421	0,38630	0,02	2,0550	0,51375
	8	0,0228	2,47368	0,30921	0,015	2,74	0,34250
	16	0,0215	2,62326	0,16395	0,0094	4,37234	0,27327
rmat-er	1	3,7992	1	1	1,3912	1	1
	2	2,5532	1,48802	0,74401	0,963	1,44465	0,72233
	4	1,3729	2,76728	0,69182	0,6521	2,13342	0,53335
	8	0,7694	4,93787	0,61723	0,3785	3,67556	0,45945
	16	0,611	6,21800	0,38863	0,2624	5,30183	0,33136
rmat-b	1	2,5821	1	1	0,5273	1	1
	2	2,0021	1,28970	0,64485	0,3472	1,51872	0,75936
	4	1,1552	2,23520	0,55880	0,2945	1,79049	0,44762
	8	0,6392	4.03958	0,50495	0,1883	2,80032	0,35004
	16	0,415	6.22193	0,38887	0,1358	3,88292	0,24268
europe	1	1,6455	1	1	1,51770	1	1
	2	0,9164	1,79561	0,89781	1,3297	1,14139	0,57069
	4	0,8339	1,97326	0,49331	0,8813	1,72212	0,43053
	8	0,6363	2,58604	0,32326	0,6642	2,28500	0,28563
	16	0,7946	2,07085	0,12943	0,794	1,91146	0,11947
wiki	1	0,2651	1	1	0,0797	1	1
	2	0,1447	1,83207	0,91603	0,0783	1,01788	0,50894
	4	0,084	3,15595	0,78899	0,0406	1,96305	0,49076
	8	0,0516	5,13760	0,64220	0,0267	2,98502	0,37313
	16	0,105	2,52476	0,15780	0,0486	1,63992	0,10249

Table 1: Multicore-CPU results

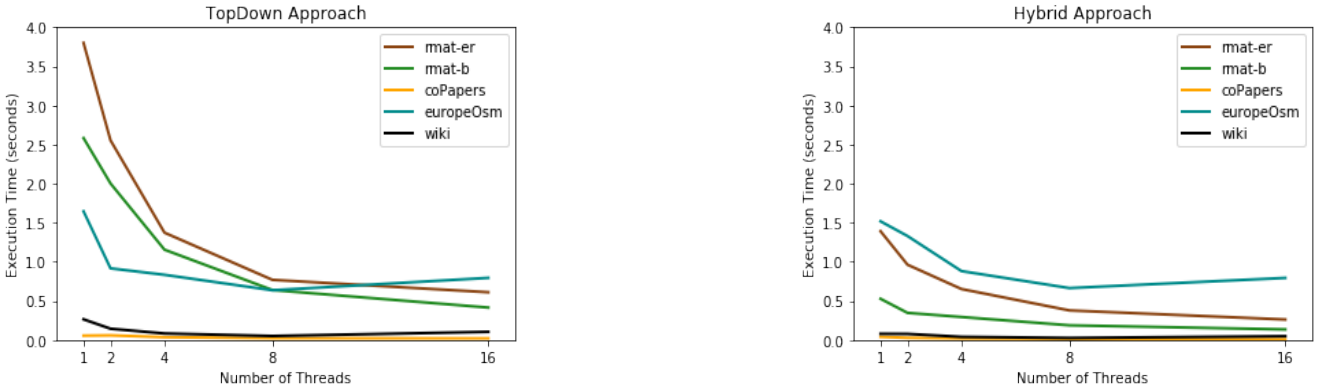

Figure 2: Execution Time Comparison: Top-Down vs Hybrid

- Stunningly serial execution times are lowered using the hybrid approach as well.

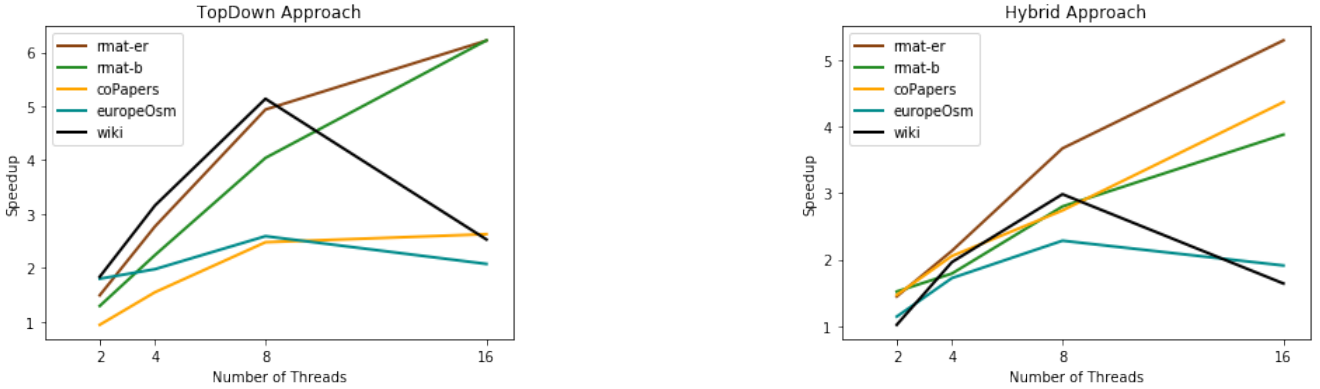

Figure 3: Speedup Comparison: Top-Down vs Hybrid

- With the hybrid approach, the speedup of coPapers graph is increasing even after 8 threads, proving that the Hybrid approach may earn more speedup with increasing number of threads.

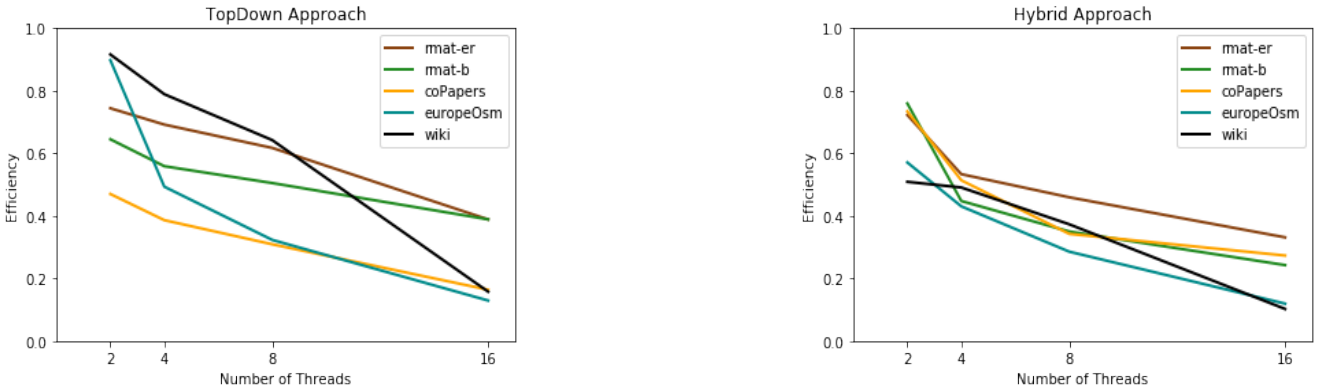

Figure 4: Efficiency Comparison: TopDown vs Hybrid

- Although Hybrid approach generally works better, efficiency results in higher values with the Top-Down approach.

6.2 Nvidia GPU Results

Note that the corresponding serial execution (TopDown to TopDown and Hybrid to Hybrid) was taken into account while calculating speedup and efficiency. In addition, europeOsm's results were shown separately because of its slower execution times. Table 2 & 3 provides execution times and speedup values of GPU implementations on 5 various graph structues as in OpenMP results. Figure 5 & 6 depict execution and speedup values of results in Table 2 & 3 in a better visual format. GPU architecture information are depicted below:

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)

Level-based GPU CUDA Implementation				
Graph	TopDown(s)	TD (Speedup)	Hybrid (s)	H (Speedup)
coPapers	0,055	1,02545	0,00360	11,41667
rmat-er	0,1671	22,73609	0,02670	52,10487
rmat-b	0,1738	14,85763	0,15960	3,30388
europe	10,7791	0,15266	10,73510	0,14138
wiki	0,0322	8,23292	0,02310	3,45022

Table 2: Level-based GPU CUDA results

Frontier-based GPU CUDA Implementation				
Graph	TopDown(s)	TD (Speedup)	Hybrid (s)	H (Speedup)
coPapers	0,00960	5,875	0,00440	9,34091
rmat-er	0,29470	12,89175	0,11	12,64727
rmat-b	0,22680	11,38492	0,07480	7,04947
europe	0,52780	3,11766	0,58730	2,58420
wiki	0,03320	7,98494	0,01990	4,00503

Table 3: Frontier-based GPU CUDA results

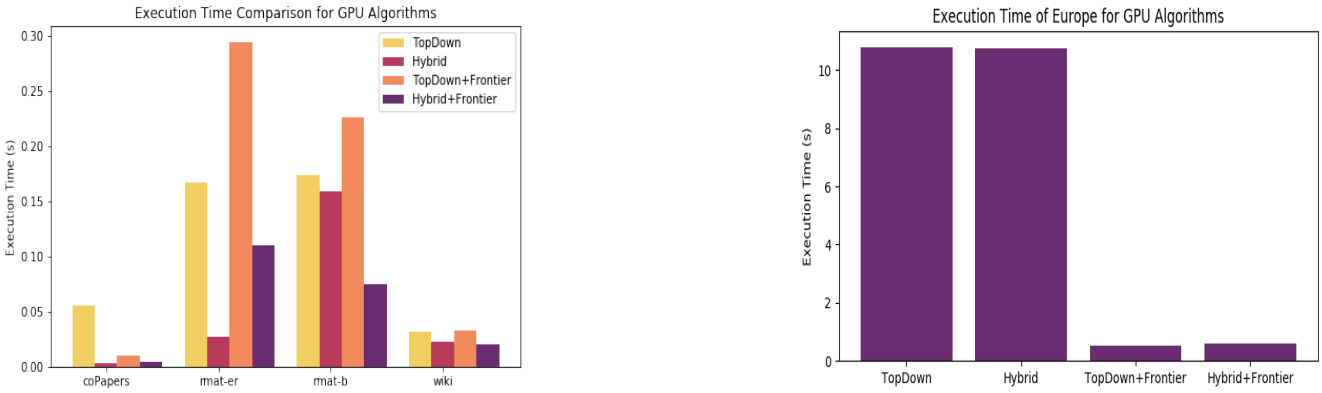

Figure 5: Execution Time Comparison

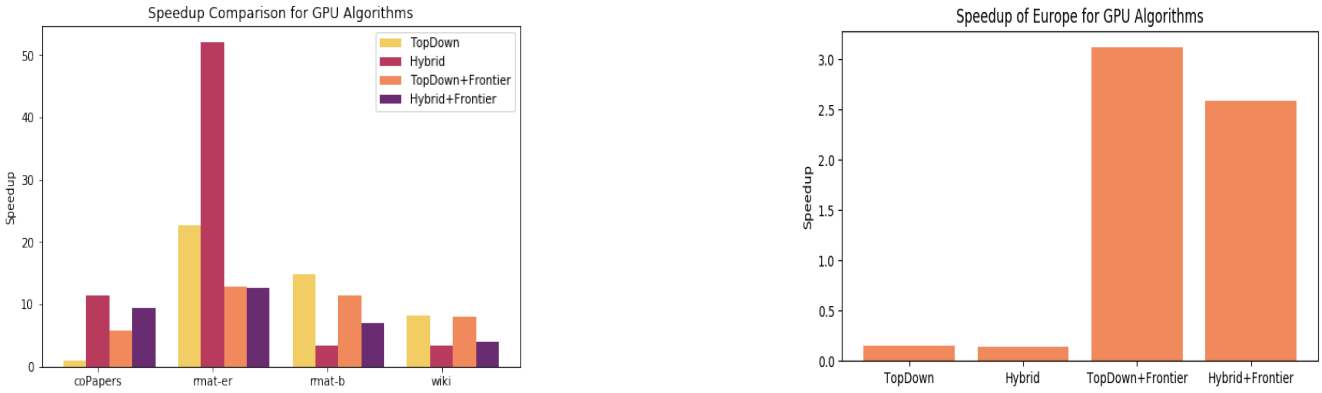

Figure 6: Speedup Comparison

Frontier approach works very well when there are many levels and small number of nodes per level. For europeOsm graph; since there are very many frontiers, layer-based approach performs extremely bad. It loops over all the edges for every frontier. However, frontier approach performs atomic operations which are costly on GPUs. Thus; for rmat-er, as it is assumed there are not very many levels, the layer-based approach performs much better than the frontier approach. Regardless of approach and graph, it can be seen that hybrid approach performs always better than the TopDown approach.

6.3 Comparison between Multicore CPU and GPU

Note that the corresponding serial execution (TopDown to TopDown and Hybrid to Hybrid) was taken into account while calculating speedup and efficiency. Figure 7 demonstrate speedup value comparisons of multicore-CPU and GPU implementations that are provided in previous subsections, namely Table 1, 2 & 3 as well as Figure 2, 3, 4 & 5.

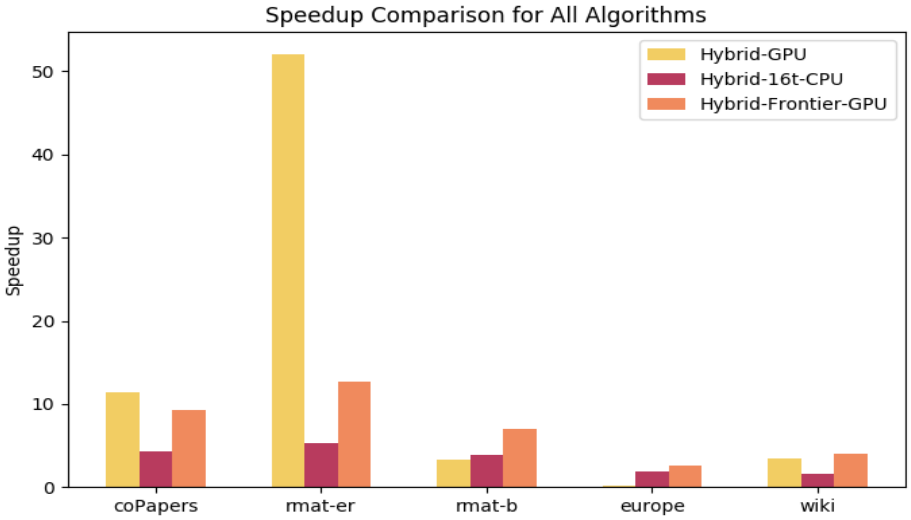

Figure 7: Speedup Comparison of All Algorithms

7 Appendix

```
int size = 1;
results[startNode] = 0;
mrQ[0] = startNode;
while(size)
{
  int ctr = 0;
#pragma omp parallel num_threads(t)
  {
    int* temp = tempArr[omp_get_thread_num()];
    int tempCtr = 0;
    int sum = 0;
    int i, v, end;
#pragma omp for reduction(+:ctr) schedule(guided)
    for(int j = 0; j < size; j++)
      {
   v = front[j];
   i = start_ind[v];
   end = start_ind[v+1];
   for(i; i < end; i++)
     {
       int index = target_ind[i];
       if(results[index] == -1)
         {
      results[index] = results[v]+1;
      temp[tempCtr++] = index;
      ctr++;
         }
     }
      }
    int tid = omp_get_thread_num();
    prefixSum[tid] = tempCtr;
#pragma omp barrier
    size = ctr;
    for(int m = 0; m < tid; m++)
      sum += prefixSum[m];
    for(int k = 0 ; k < tempCtr; k++)
      front[sum++] = temp[k];
  }
```

Appendix 1: Top-Down BFS Implementation using OpenMP

```
int size = 1;
results[startNode] = 0;
mrQ[0] = startNode;
while(size)
{
    int ctr = 0;
#pragma omp parallel num_threads(t)
  {
    int* temp = tempArr[omp_get_thread_num()];
    int tempCtr = 0;
    int sum = 0;
#pragma omp for reduction(+:ctr) schedule(guided)
    for(int ii = 0; ii < RS; ii++)
      {
   int st, end;
   if(results[ii] == -1)//check if it has any distance(parent)
     {
       st = start_ind[ii];
       end = start_ind[ii+1];
       bool raviolli = false;
       for(int jj = st; jj < end && raviolli == false; jj++)
         {
      int target = target_ind[jj];//target is a neighbor of v
      if(front_check[target])//if
        {
          results[ii] = results[target] + 1;
          temp[tempCtr++] = ii;
          ctr++;
          raviolli = true;
        }
         }
     }
      }
    int tid = omp_get_thread_num();
    prefixSum[tid] = tempCtr;
#pragma omp barrier
    size = ctr;
    for(int m = 0; m < tid; m++)
      sum += prefixSum[m];
    for(int k = 0 ; k < tempCtr; k++)
      front[sum++] = temp[k];

  }
 }
```

Appendix 2: Bottom-Up BFS Implementation using OpenMP

```
__global__
void BFS_Top_Down(int* target_ind, int* start_ind, int* results, int v, int RS,
unsigned int TOTAL, int* SIZE)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;


  for(int jj = index; jj < RS; jj+=TOTAL)
    {
      if(results[jj] == v)
       {
           atomicAdd(SIZE,1);
           int start_loc = start_ind[jj];
           int end_loc = start_ind[jj + 1];
           int curr = results[jj];
           for(int ii = start_loc; ii < end_loc; ii++)
             {
               if(results[target_ind[ii]] == -1)
            {
               results[target_ind[ii]] = curr + 1;
               }
            }
        }
     }
}
```

Appendix 3: Top-Down BFS Implementation using CUDA(Layer-Based)

```
__global__
void BFS_Bottom_Up(int* target_ind, int* start_ind, int* results, int v, unsigned int RS,
unsigned int TOTAL, int* SIZE)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;

  for(int jj = index; jj < RS; jj+=TOTAL)
    {
      if(results[jj] == -1)
       {
       int start_loc = start_ind[jj];
       int end_loc = start_ind[jj + 1];
       int target;
       bool raviolli = false;
       for(int ii = start_loc; ii < end_loc && raviolli == false; ii++)
           {
           target = results[target_ind[ii]];
           if(target == v)
          {
              results[jj] = target + 1;
              atomicAdd(SIZE, 1);
              raviolli = true;
          }
        }
     }
   }
}
```

Appendix 4: Bottom-Up BFS Implementation using CUDA(Layer-Based)

```
__global__
void BFS_Top_Down(int* target_ind, int* start_ind, int* results, int* frontier, int* frontsize,
int* newfrontier, int* newfrontsize, int RS)
{
  int p = blockIdx.x * blockDim.x + threadIdx.x;

  if(p < (*frontsize))
    {
      int index = frontier[p];
      int start_loc = start_ind[index];
      int end_loc = start_ind[index + 1];
      int curr = results[index];
      for(int ii = start_loc; ii < end_loc; ii++)
       {
       if(results[target_ind[ii]] == -1)
           {
           results[target_ind[ii]] = curr + 1;
           int cc = atomicAdd(newfrontsize,1);
           newfrontier[cc] = target_ind[ii];

           }
       }
    }
}
```

Appendix 5: Top-Down BFS Implementation using CUDA(Frontier-Based)

```
__global__
void BFS_Bottom_Up(int* front_check, int* target_ind, int* start_ind, int* results, int *↩
    frontier, int* frontsize, int* newfrontier, int* newfrontsize, unsigned int RS, unsigned int↩
     TOTAL)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;

  for(int j = index; j < RS; j += TOTAL)
    {
      if(results[j] == -1)
       {
       int start_loc = start_ind[j];
       int end_loc = start_ind[j + 1];
       int target;
       int raviolli = 0;
       for(int ii = start_loc; ii < end_loc && raviolli == 0; ii++)
       {
         target = target_ind[ii];
         if(front_check[target])
          {
          results[j] = results[target] + 1;
          int cc = atomicAdd(newfrontsize, 1);
          newfrontier[cc] = j;
          raviolli = 1;
          }
       }
    }
  }
}
```

Appendix 6: Bottom-Up BFS Implementation using CUDA(Frontier-Based)

References

[1] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 12:1–12:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[2] M. Belova and M. Ouyang. Breadth-first search with a multi-core computer. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 579–587, May 2017.

[3] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 117–128, New York, NY, USA, 2012. ACM.