Graph Algorithms Framework in SYCL

Prasanna Bartakke (EE19B106), guided by Prof. Rupesh Nasre

December 1, 2022

1 Introduction

1.1 Parallelising Graph Algorithms

Many problems in the real world can be modelled as graph algorithms, for example, social networks, road connections, molecular interactions, and planetary forces. The execution time required for these algorithms is considerable for large graphs. They can be parallelised and sped up using GPUs. Once an algorithm is implemented for a specific backend, e.g. NVIDIA GPUs in CUDA, it is not possible to execute the same implementation of the algorithm on other GPUs like AMD or Intel.

1.2 Objectives

- To enable the platform portability of graph algorithms by implementing them in SYCL.
- To develop a framework that enables implementing graph algorithms quickly, hiding the complexities of SYCL syntax and adding commonly used functionalities related to graph algorithms to the framework.
- To implement the following graph algorithms: single source shortest path, betweenness centrality, pagerank, minimum spanning tree, and triangle count using the developed framework and comparing their performance on large graphs with the existing parallel implementations[1].

1.3 SYCL

SYCL is a single-source, high-level, standard C++ programming model that can target a range of heterogeneous platforms. SYCL implementations are available from various vendors. Intel's implementation of SYCL, DPC++, has been used in this project. It is based on LLVM/Clang and is a part of oneAPI. DPC++ code can run on any CPU, Intel, NVIDIA and AMD GPUs.

[1]The LoneStarGPU[8] and Gunrock[7] implementations are used for comparison. The execution time for existing implementations was obtained by other StarPlat members as part of a larger project.

2 Framework

Users can input raw graphs represented as an edge list in a text file. The raw graphs are converted into the Compressed Sparse Row(CSR) representation. The conversion step is done only once for a given graph, and the CSR representation can be used in multiple algorithms. While converting the raw graph into its CSR representation, the self-loops and duplicate edges between two vertices are removed, and the vertex ids are continuous. Also, each edge is randomly assigned a weight between 1 and 100. The directed graphs can be converted to undirected graphs by specifying the relevant parameters while executing the script. The neighbours of all vertices are stored in a sorted order to optimise the time required to find if two vertices are connected using binary search. All the data related to the graph, like the number of vertices, the number of edges and the CSR arrays(`V, I, E, W, RE, RI`), are stored as properties of the graph object. The CSR arrays are stored on the device, and their pointers are in the graph object. The graph object is present in the Unified Shared Memory(USM), which can be accessed from both the host and the device. This method for memory allocations avoids data transfer between the host and the device every time a new kernel is executed.
Following are the functions that can be used in the implementations of different algorithms to get the graph-related data without explicitly accessing the CSR arrays: `get_node(j)`, `get_neighbour(j)`, `get_parent(j)`, `get_weight(j)`, `get_num_neighbours(j)`, `begin_neighbours(u)`, `end_neighbours(u)`, `begin_parents(u)`, `end_parents(u)`.
The framework provides two iterators, `for_neighbours(u, j)`, `for_parents(u, j)`, to iterate over the neighbours and parents of a vertex, respectively. The framework also provides the iterator `forall(N, NUM_THREADS)` to run `NUM_THREADS` work items in parallel. Some other functions in

the framework are initialising memory, copying memory to/from the device to the host and checking if two vertices are connected using binary search.

3 Graph Algorithms

Following are some of the challenges we have tried to solve in parallelising graph algorithms using the developed framework:

- Avoiding atomic instructions as they quickly become expensive on GPUs.
- Balancing the work done by each thread as it varies considerably depending on the graph structure.
- Exploiting memory locality and avoiding the frequent transfer of memory from the host to the device.

All graph algorithms are tested on ten large graphs with the following properties:

Table 1: Test Graph properties

GRAPH	num vertices	num edges
wikipedia-link	3370462	93373056
soc-pokec	1632803	30622564
soc-twitter-2010	21297772	265025809
soc-sinaweibo	58655849	261321071
com-Orkut	3072626	234370166
soc-LiveJournal1	4847571	68993773
USARoadNet	23947347	57708624
RMAT	16775681	87654320
Uniform Random	10000000	80000000
GermanyRoadNet	11548845	24738362

3.1 Single source shortest path

This problem involves finding the shortest paths between a given source vertex `v` and all other vertices in the graph. The Bellman-ford algorithm is parallelised[1] using topology-driven and data-driven methods. The Bellman-ford algorithm uses the edge relaxation technique. Edge relaxation is a method to correct the approximate distances with the correct ones.

Algorithm 1: Edge Relaxation

```
func edge_relaxation(v):
    if $(d[u] + w < d[v])$ then
        d[v] = d[u] + w;
    end
```

Let `d[u]` be the distance between the source and vertex `u`, and `d[v]` be the distance between the source and vertex `v`. There is an edge between `u` and `v` with weight `w`. If the distance of `v` is more than the distance of `u` + `w`, then the distance of `v` is updated with `d[u] + w`.

Algorithm 2: SSSP Operator

```
func sssp_operator(u):
    for $v$ in neighbour(u) do
        atomic edge_relaxation(v);
    end
```

In the topology-driven method, the `sssp_operator` is applied in parallel to all the nodes. In the data-driven method, the `sssp_operator` is applied to nodes where work needs to be done in parallel. This is achieved by maintaining a worklist of the active nodes. The topology-driven method performs extra work, whereas the data-driven approach is work-efficient. The drawback of the data-driven approach is the need for a centralized worklist that requires atomic insertions.
The execution time(in seconds) of our implementations(DD, TD) on NVIDIA GPU is compared with the existing implementations from the latest papers. The same implementation was also executed on Intel GPU and CPU. The NVIDIA GPU outperforms Intel GPU; however, we achieve reasonable speedup compared to sequential implementations on CPU on Intel GPUs.

Table 2: SSSP execution time in seconds

Graph	DD	TD	LoneStar	Gunrock	OpenACC
wiki	0.10	0.10	0.06	0.56	0.69
pokec	0.03	0.02	0.04	0.31	0.48
twitter	0.24	0.22	0.05	2.27	1.60
sinaweibo	0.24	0.17	0.08	4.06	3.19
orkut	0.11	0.09	0.22	0.62	1.09
liv	0.05	0.03	0.08	0.56	0.67
USA	8.69	5.50	0.16	1.28	3.26
rmat	0.11	0.09	0.13	1.03	0.75
u10m_80m	0.15	0.12	0.18	0.92	-
germany	5.54	1.86	0.09	1.14	2.12

The time-driven version outperforms the data-driven version in all graphs as the overhead of atomic insertion in the worklist is higher than the inefficient work across different threads.

3.2 Pagerank

The Pagerank algorithm[5] measures the importance of each node within the graph based on the number of incoming relationships and the corresponding source nodes' importance.
The input to the algorithm is a graph that can have spider traps and dead ends and a parameter β. The output is the pagerank vector r^{new}. The following steps are repeated until convergence: $\sum_j |r_j^{new} - r_j^{old}| < diff$

- $\forall j : r_j'^{new} = \sum_{i \to j} \beta \frac{r_i^{old}}{d_i}$

- Now, re-insert the leaked Pagerank: $\forall j : r_j^{new} = r_j'^{new} + \frac{1-\beta}{N}$
- $r^{old} = r^{new}$

Each node updates its pagerank value by looking at its parents. The pull-based approach does not require atomics, as race conditions will not happen.

Table 3: Pagerank execution time in seconds

Graph	SYCL-I	SYCL-N	LoneStar	Gunrock
wiki	28.94	8.12	0.104	2.46
pokec	13.29	1.39	0.24	1.085
twitter	92.36	27.57	-	15.23
sinaweibo	136.94	15.60	0.24	36.91
orkut	44.34	9.66	0.363	2.43
liv	23.50	2.42	0.225	2.952
USA	28.41	4.10	0.832	13.345
rmat	29.40	9.29	0.24	9.17
u10m_80m	113.76	10.17	0.24	5.487
germany	20.12	2.00	0.294	6.499

The above table compares the execution time of our implementation of the algorithm in SYCL on NVIDIA(SYCL-N) and Intel(SYCL-I) GPUs and existing CUDA implementations. Our implementation performs worse than LoneStar; however, it performs similarly to Gunrock.

3.3 Triangle count

The Triangle Count algorithm counts the number of triangles for each node in the graph. A triangle is a set of three vertices where each vertex is connected to the other two. In graph theory terminology, this is sometimes referred to as a 3-clique.

Algorithm 3: TC operator

```
func tc_operator(u):
    for v in neighbour(u) do
        for w in neighbour(u) do
            if (v < u < w) and is_connected(v, w) then
                atomic increment triangle count;
            end
        end
    end
```

The `tc_operator` is invoked in parallel for all the nodes. We iterate over all the distinct pairs of neighbours for a given node and increment the triangle count atomically if those nodes are connected. As the neighbours are stored in sorted order in E, the `is_connected` function is optimised using binary search.

Table 4: Triangle count execution time in seconds

Graph	num triangles	SYCL-I	SYCL-N	CUDA
pokec	32,557,458	3.80	1.03	0.66
orkut	627,584,181	107.20	40.60	46.70
liv	285,730,264	23.36	3.23	3.01
USA	438,804	5.50	4.32E-03	1.54E-03
u10m_80m	665	2.63	0.35	1.04
germany	13,390	0.09	2.64E-03	7.00E-04
rmat	1,968,238,446	3,001.07	2,311.25	2567.57
wiki	1,977,659,897	11,659.38	8,482.19	7,992.96

3.4 Betweenness Centrality

Betweenness Centrality is a measure of the influence of a vertex in a graph. It measures the ratio of shortest paths passing through a particular vertex to the number of shortest paths between all pairs of vertices. Intuitively, this ratio determines how well a vertex connects pairs of vertices in the network. Mathematically, the betweenness centrality of a vertex `v` is defined as
$BC(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$, where σ_{st} is the number of shortest paths between vertices `s` and `t`, and $\sigma_{st}(v)$ is the number of those paths that pass through `v`.
The naive implementation of solving the all-pairs shortest paths problem is of the order of O(V^3) by the Floyd-Warshall technique, and betweenness centrality is calculated by counting the necessary shortest paths. Brandes algorithm aims to reuse the already calculated shortest distances by using partial dependencies of shortest paths between pairs of nodes for calculating the betweenness centrality of a given vertex w.r.t. a fixed source. A recursive relation for defining the partial dependency(δ) w.r.t. source `s` is:
$\delta_s(v) = \sum_{w \in succ(v)} \frac{\sigma_{sv}}{\sigma_{sw}}(1 + \delta_s(w))$
$BC(v) = \sum_{s \neq v} \delta_s(v)$
Effectively the calculation of betweenness centrality values can then be divided into 2 steps:
1. Find the shortest paths between each pair of vertices: This is done by fixing a source and doing a forward breadth-first search to calculate the depth of other vertices w.r.t. the source.
2. Sum the dependencies for each vertex: This is done using a reverse breadth-first search by level-order traversal from the deepest level towards the source. The partial dependencies of the shortest paths between the current vertex to the fixed source are calculated, passing for all the predecessor vertices(the vertices which are immediately one level above the current vertex) and the shortest path from the source to the current element passes through them. This gives a run time complexity of O($V \times E$).
The vertex-parallel approach[6] assigns a thread to each vertex of the graph and traverses all outgoing edges from that vertex. For both the shortest path calculation and the dependency accumulation stages, the number of edges traversed per-thread by the vertex-parallel approach depends on the out-degree of the vertex assigned to each thread. The difference in out-degrees between vertices causes a load imbalance between threads.
In the work-efficient approach[2], explicit queues

are used for graph traversal. Since levels of the graph are processed in parallel, we use two queues to distinguish vertices in the current level of the search (`q_curr`) from vertices that are to be processed during the next level of the search (`q_next`). For the dependency accumulation stage, we initialize `S` and its length. In this case, we need to keep track of vertices at all levels of the search, and hence we use only one data structure to store these vertices. We use the `ends` array to distinguish the sections of `S` that correspond to each level of the search. This usage of the ends and `S` arrays is analogous to the arrays used to store the graph in CSR format. The atomic `compare_exchange_next` instruction prevents multiple insertions of the same vertex into `q_next`.

The following table compares the execution time of the BC algorithm for a single source in SYCL(both vertex parallel and work efficient methods) on NVIDIA GPU and the existing CUDA implementation. We observe that the execution time increases linearly with the number of sources for all graphs.

Table 5: BC execution time in seconds

Graph	SYCL-VP	SYCL-WE	LoneStar	Gunrock
wiki	0.38	0.59	0.155	0.535
pokec	0.05	0.11	0.03	0.317
twitter	0.95	1.25	-	2.122
sinaweibo	0.60	0.96	-	4.237
orkut	0.17	0.31	0.149	0.525
liv	0.09	0.18	0.073	0.548
USA	6.41	1.93	17.912	2.811
rmat	0.37	0.51	0.211	1.238
u10m_80m	0.20	0.55	0.08	0.944
germany	4.02	1.00	6.485	1.75

We can see that the execution time of the work efficient method is higher than the vertex parallel method for all the graphs except the road networks(USA, Germany). This may happen due to the lower ratio of the number of edges to the number of vertices in the road networks compared to other graphs.

3.5 Minimum Spanning Tree

A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together without any cycles and with the minimum possible total edge weight. Traditional MST algorithms like Kruskal's and Prim's algorithms are inherently sequential. Boruvka's MST algorithm is another well-known algorithm that can be parallelised.

Boruvka's algorithm begins by finding the min-weight edge from each vertex and adding all these to the Minimum Spanning Forest(MSF). Then the min-weight edge from each tree in the forest to a different tree is found, ensuring that no cycles are formed and adds it to the MSF. This is repeated until no more edges can be added.

Following are the steps[3] to parallelise MST using Boruvka's algorithm:

Algorithm 4: MST steps

```
while single component not found do
    1. find MST edges;
    2. remove cycles;
    3. update MST edges;
    4. update parents;
    5. flatten parents;
end
```

The first step, finding MST edges, is split into two sub-steps as follows:

- Finding the minimum weight edge from each vertex using the topology-driven method.
- Finding the minimum weight edge from each component. Similar to the above step, each vertex is processed by a different thread. Multiple vertices belonging to a particular component try to update the minimum weight edge of that component, which leads to a data race. This step is run until the component's correct minimum weight edge is not found. Convergence is guaranteed in at most `c_i` steps for each component, where `c_i` is the number of vertices in the i^{th} component. This is because the number of vertices causing data race is reduced by at least one in each step.

In the above steps, if two edges have the same weight, the node with the smaller parent id is selected. This prevents cycles of greater than two vertices from being formed. The cycles of length two are removed in the next step. The newly selected edges are updated in the MST, and the vertices' parents are updated. All the vertices present in a component have the same parent; this is done in the flatten parents step.

Table 6: MST execution time in seconds

Graph	MST edges	SYCL-I	SYCL-N	CUDA
wiki	3,370,376	3.07	0.61	0.72
pokec	1,632,802	0.81	0.13	0.19
twitter	16,061,344	8.19	2.49	3.22
sinaweibo	43,785,779	7.97	0.90	2.01
orkut	3,072,440	4.03	0.83	0.64
liv	4,845,695	1.71	0.21	0.23
USA	23,947,346	2.66	0.16	0.17
rmat	5,392,997	3.72	0.72	0.92
u10m_80m	9,999,998	5.55	0.87	0.58
germany	11,548,844	1.17	0.10	0.01

The above table compares the execution time of our implementation of the algorithm in SYCL on NVIDIA and Intel GPUs and the CUDA implementation[4]. Our implementation has better execution time in all the graphs except the road networks.

4 Concluding Remarks

- We have used state-of-the-art parallelisation techniques for optimizing each algorithm and implemented them using the developed framework. The developed framework can be utilised for implementing custom applications involving graph algorithms.
- These algorithms have been tested on three devices (2GPUs + 1CPU), and their performance is comparable to the existing parallel implementations from the latest papers.

5 Acknowledgements

- I would like to thank Prof. Rupesh Nasre for his guidance and support throughout the semester, without whom I would not have been able to complete this project.
- I am grateful to Rajesh Pandian for patiently helping me to install SYCL on NVIDIA GPU.
- I would also like to thank Vineeth Kada for helping me with the convergence method in MST.

References

[1] F. Busato and N. Bombieri, ”An Efficient Implementation of the Bellman-Ford Algorithm for Kepler GPU Architectures,” in IEEE Transactions on Parallel and Distributed Systems, vol. 27, no. 8, pp. 2222-2233, 1 Aug. 2016, doi: 10.1109/TPDS.2015.2485994.

[2] Adam McLaughlin and David A. Bader. 2014. Scalable and high performance betweenness centrality on the GPU. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC ’14). IEEE Press, 572–583. https://doi.org/10.1109/SC.2014.52

[3] Vineet, V., P. Harish, S. Patidar, and P. Narayanan (2009). Fast minimum spanning tree for large graphs on the gpu. In Proceedings of the Conference on High Performance Graphics 2009, pp. 167–171.

[4] A. Mariano, A. Proenca and C. Da Silva Sousa, ”A Generic and Highly Efficient Parallel Variant of Boruvka’s Algorithm,” 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 2015, pp. 610-617, doi: 10.1109/PDP.2015.72.

[5] Sergey Brin, Lawrence Page,The anatomy of a large-scale hypertextual Web search engine,Computer Networks and ISDN Systems, Volume 30, Issues 1–7,1998,Pages 107-117,ISSN 0169-7552,https://doi.org/10.1016/S0169-7552(98)00110-X.

[6] Jia, Y., Lu, V., Hoberock, J., Garland, M., Hart, J.C. Edge v. node parallelism for graph centrality metrics. GPU Computing Gems 2 (2011), 15–30.

[7] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. 2017. Gunrock: GPU Graph Analytics. ACM Trans. Parallel Comput. 4, 1, Article 3 (March 2017), 49 pages. https://doi.org/10.1145/3108140

[8] A Quantitative Study of Irregular Programs on GPUs Martin Burtscher, Rupesh Nasre, Keshav Pingali IEEE International Symposium on Workload Characterization (IISWC) 2012

Appendix

Single source shortest path

Topology driven version

```
int *dev_flag = malloc_device<int>(N, Q);
initialize(dev_flag, 0, NUM_THREADS, N, Q, source, 1);
int *dev_dist = malloc_device<int>(N, Q);
initialize(dev_dist, INT_MAX, NUM_THREADS, N, Q, source, 0);

int *dev_dist_i = malloc_device<int>(N, Q);
initialize(dev_dist_i, INT_MAX, NUM_THREADS, N, Q, source, 0);

int *early_stop = malloc_shared<int>(1, Q);
for (int round = 1; round < N; round++)
{
    if (*early_stop) break;

    *early_stop = 1;
    forall(N, NUM_THREADS)
    {
        if (dev_flag[u])
        {
            dev_flag[u] = 0;
            int u_itr;
            for_neighbours(u, u_itr)
            {
                int v = get_neighbour(u_itr);
                int w = get_weight(u_itr);

                int du = dev_dist[u];
                if (du == INT_MAX) continue;

                int dv = dev_dist[v];
                int new_dist = du + w;

                ATOMIC_INT atomic_data(dev_dist_i[v]);
                atomic_data.fetch_min(new_dist);
            }
        }
    }
    end;

    forall(N, NUM_THREADS)
    {
        if (dev_dist[u] > dev_dist_i[u])
        {
            dev_dist[u] = dev_dist_i[u];
            dev_flag[u] = 1;
            *early_stop = 0;
        }
        dev_dist_i[u] = dev_dist[u];
    }
    end;
}
```

Data driven version

```
int *dev_dist = malloc_device<int>(N, Q);
initialize(dev_dist, INT_MAX, NUM_THREADS, N, Q, source, 0);

int *dev_dist_i = malloc_device<int>(N, Q);
initialize(dev_dist_i, INT_MAX, NUM_THREADS, N, Q, source, 0);

int *active_count = malloc_shared<int>(1, Q);
int *wl = malloc_shared<int>(N, Q);
*active_count = 1;
wl[0] = source;

for (int round = 1; round < N; round++)
{
    if (*active_count == 0) break;

    forall(*active_count, NUM_THREADS)
    {
        int node = wl[u];
        int n_itr;
        for_neighbours(node, n_itr)
        {
            int w = get_weight(n_itr);
            int v = get_neighbour(n_itr);

            int dnode = dev_dist[node];
            int dv = dev_dist[v];
            int new_dist = dnode + w;

            if (dnode == INT_MAX) continue;

            ATOMIC_INT atomic_data(dev_dist_i[v]);
            atomic_data.fetch_min(new_dist);
        }
    }
    end;

    *active_count = 0;
    free(wl, Q);
    wl = malloc_shared<int>(N, Q);

    forall(N, NUM_THREADS)
    {
        if (dev_dist[u] > dev_dist_i[u])
        {
            dev_dist[u] = dev_dist_i[u];
            ATOMIC_INT atomic_data(*active_count);
            wl[atomic_data++] = u;
        }
        dev_dist_i[u] = dev_dist[u];
    }
    end;
}
```

Pagerank

```
float *dev_pagerank = malloc_device<float>(N, Q);
initialize(dev_pagerank, float(1 / N), NUM_THREADS, N, Q);

float *dev_pagerank_i = malloc_device<float>(N, Q);
initialize(dev_pagerank_i, float(1 / N), NUM_THREADS, N, Q);

float *diff = malloc_shared<float>(1, Q);
int iterCount = 0;
do
{
    forall(N, NUM_THREADS)
    {
        float sum = 0;
        // pull
        int u_itr;
        for_parents(u, u_itr)
        {
            int nbr = get_parent(u_itr);
            sum = sum + dev_pagerank[nbr] / (get_num_neighbours(nbr));
        }
        float val = (1 - delta) / N + delta * sum;
        ATOMIC_FLOAT atomic_data(*diff);
        atomic_data += (float)val - dev_pagerank[u];
        dev_pagerank_i[u] = val;
    }
    end;

    forall(N, NUM_THREADS)
    {
        dev_pagerank[u] = dev_pagerank_i[u];
    }
    end;

    iterCount += 1;
} while ((*diff > beta) && (iterCount < maxIter));
```

Triangle Count

```
int triangle_count = 0;
int *dev_triangle_count = malloc_device<int>(1, Q);

memcpy(dev_triangle_count, &triangle_count, 1, Q);

forall(N, NUM_THREADS)
{
    int u_itr1, u_itr2;
    for_neighbours(u, u_itr1)
    {
        int v = get_neighbour(u_itr1);
        if (v < u)
        {
            for_neighbours(u, u_itr2)
            {
                int w = get_neighbour(u_itr2);
                int nbrs_connected = 0;
                if (w > u) nbrs_connected = neighbours(v, w, g);

                if (nbrs_connected)
                {
                    ATOMIC_INT atomic_data(dev_triangle_count[0]);
                    atomic_data += 1;
                }
            }
        }
    }
}
end;
```

Betweenness Centrality

Vertex parallel version

```
float *dev_bc = malloc_device<float>(bc.size(), Q);
initialize(dev_bc, float(0), NUM_THREADS, N, Q);

float *dev_delta = malloc_device<float>(N, Q);
int *dev_d = malloc_device<int>(N, Q);
int *dev_sigma = malloc_device<int>(N, Q);

int *S = malloc_shared<int>(N, Q);
int *ends = malloc_shared<int>(N, Q);
int *position = malloc_shared<int>(1, Q);
int *s = malloc_shared<int>(1, Q);
```

```
*s = 0;
int *finish_limit_position = malloc_shared<int>(1, Q);
int *done = malloc_shared<int>(1, Q);
int *current_depth = malloc_shared<int>(1, Q);

for (auto x : sourceSet)
{
    *s = x;
    initialize(dev_d, INT_MAX, NUM_THREADS, N, Q, *s, 0);
    initialize(dev_sigma, 0, NUM_THREADS, N, Q, *s, 1);
    initialize(dev_delta, float(0), NUM_THREADS, N, Q);

    *current_depth = 0;
    *done = 0;
    *position = 0;
    *finish_limit_position = 1;
    ends[0] = 0;

    while (!*done)
    {
        *done = 1;
        forall(N, NUM_THREADS)
        {
            if (dev_d[u] == *current_depth)
            {
                ATOMIC_INT atomic_data(*position);
                int t = atomic_data++;
                S[t] = u;
                int r;
                for_neighbours(u, r)
                {
                    int w = get_neighbour(r);
                    if (dev_d[w] == INT_MAX)
                    {
                        dev_d[w] = dev_d[u] + 1;
                        *done = 0;
                    }

                    if (dev_d[w] == (dev_d[u] + 1))
                    {
                        ATOMIC_INT atomic_data(dev_sigma[w]);
                        atomic_data += dev_sigma[u];
                    }
                }
            }
        }
        end;
        *current_depth += 1;
        ends[*finish_limit_position] = *position;
        ++*finish_limit_position;
    }

    for (int itr1 = *finish_limit_position - 1; itr1 >= 0; itr1--)
    {
        forall(ends[itr1 + 1] - ends[itr1], NUM_THREADS)
        {
            int itr2 = u + ends[itr1];
            int v = S[itr2];
            int itr3;
            for_neighbours(v, itr3)
            {
                int w = get_neighbour(itr3);
                if (dev_d[w] == dev_d[v] + 1)
                {
                dev_delta[v] += ((dev_sigma[v] / dev_sigma[w]) * (1 + dev_delta[w]));
                }
            }

            if (v != *s) dev_bc[v] += dev_delta[v];
        }
        end;
    }
}
```

Work efficient version

```
int *dev_d = malloc_device<int>(N, Q);
int *dev_sigma = malloc_device<int>(N, Q);
float *dev_bc = malloc_device<float>(bc.size(), Q);
initialize(dev_bc, float(0), NUM_THREADS, N, Q);
float *dev_delta = malloc_device<float>(N, Q);

int *s = malloc_shared<int>(1, Q);
*s = 0;

int *q_curr = malloc_shared<int>(N, Q);
int *q_curr_len = malloc_shared<int>(1, Q);

int *q_next = malloc_shared<int>(N, Q);
int *q_next_len = malloc_shared<int>(1, Q);

int *S = malloc_shared<int>(N, Q);
int *S_len = malloc_shared<int>(1, Q);

int *ends = malloc_shared<int>(N, Q);
int *ends_len = malloc_shared<int>(1, Q);
int *depth = malloc_shared<int>(1, Q);

for (auto x : sourceSet)
{
    *s = x;
    initialize(dev_d, INT_MAX, NUM_THREADS, N, Q, *s, 0);
    initialize(dev_sigma, 0, NUM_THREADS, N, Q, *s, 1);
    initialize(dev_delta, float(0), NUM_THREADS, N, Q);

    q_curr[0] = *s;
    *q_curr_len = 1;
    *q_next_len = 0;
    S[0] = *s;
    *S_len = 1;
    ends[0] = 0;
    ends[1] = 1;
    *ends_len = 2;
    *depth = 0;

    while (1)
    {
        forall(*q_curr_len, NUM_THREADS)
        {
            int v = q_curr[u], v_itr;
            for_neighbours(v, v_itr)
            {
                int w = get_neighbour(v_itr);

                ATOMIC_INT atomic_data(dev_d[w]);
                int old = INT_MAX;
                old = atomic_data.compare_exchange_strong(old, dev_d[v] + 1);
                if (old)
                {
                    ATOMIC_INT atomic_data(*q_next_len);
                    int t = atomic_data++;
                    q_next[t] = w;
                }

                if (dev_d[w] == dev_d[v] + 1)
                {
                    ATOMIC_INT atomic_data(dev_sigma[w]);
                    atomic_data += dev_sigma[v];
                }
            }
        }
        end;

        if (*q_next_len == 0)
        {
            break;
        }
        else
        {
            forall(*q_next_len, NUM_THREADS)
            {
                q_curr[u] = q_next[u];
                S[u + *S_len] = q_next[u];
            }
            end;

            ends[*ends_len] = ends[*ends_len - 1] + *q_next_len;
            *ends_len += 1;
            int new_curr_len = *q_next_len;
            *q_curr_len = new_curr_len;
            *S_len += new_curr_len;
            *q_next_len = 0;
            *depth += 1;
        }
    }
    while (*depth >= 0)
    {
        forall(ends[*depth + 1] - ends[*depth], NUM_THREADS)
        {
            int tid = u + ends[*depth];
            int v = S[tid];
            int dsv = 0, r;
            for_neighbours(v, r)
            {
                int w = get_neighbour(r);
                if (dev_d[w] == dev_d[v] + 1)
                {
                dev_delta[v] += ((dev_sigma[v] / dev_sigma[w]) * (1 + dev_delta[w]));
                }
            }
            if (v != *s) dev_bc[v] += dev_delta[v];
        }
        end;

        *depth -= 1;
    }
}
```

Minimum Spanning Tree

```
int *d_parent = malloc_device<int>(n_vertices, Q);
copy(d_parent, g->V, NUM_THREADS, n_vertices, Q);
int *d_local_min_edge = malloc_device<int>(n_vertices, Q);
int *d_comp_min_edge = malloc_device<int>(n_vertices, Q);
int *d_edge_in_mst = malloc_device<int>(n_edges, Q);
initialize(d_edge_in_mst, 0, NUM_THREADS, n_edges, Q);

bool *rem_comp = malloc_shared<bool>(1, Q);
```

```
int *counter = malloc_shared<int>(1, Q);
bool *found_min_edge = malloc_shared<bool>(1, Q);
bool *parents_updated = malloc_shared<bool>(1, Q);

*rem_comp = false;
*counter = 0;
while (!*rem_comp)
{
    *rem_comp = true;

    initialize(d_local_min_edge, -1, NUM_THREADS, n_vertices, Q);
    initialize(d_comp_min_edge, -1, NUM_THREADS, n_vertices, Q);

    // find minimum edge to different component from each node
    forall(n_vertices, NUM_THREADS)
    {
        int edge;
        for_neighbours(u, edge)
        {
            int v = get_neighbour(edge);
            // if u and v in different components
            if (d_parent[u] != d_parent[v])
            {
                int curr_min_edge = d_local_min_edge[u];
                if (curr_min_edge == -1)
                {
                    d_local_min_edge[u] = edge;
                }
                else
                {
                    int curr_neigh = get_neighbour(curr_min_edge);
                    int curr_edge_weight = get_weight(edge), curr_min_edge_weight =
    get_weight(curr_min_edge);
                    if (curr_edge_weight < curr_min_edge_weight || (curr_edge_weight
    == curr_min_edge_weight && d_parent[v] < d_parent[curr_neigh]))
                    {
                        d_local_min_edge[u] = edge;
                    }
                }
            }
        }
    }
    end;

    // find the minimum edge from the component
    *found_min_edge = false;
    while (!*found_min_edge)
    {
        *found_min_edge = true;
        forall(n_vertices, NUM_THREADS)
        {
            int my_comp = d_parent[u];

            int comp_min_edge = d_comp_min_edge[my_comp];

            int my_min_edge = d_local_min_edge[u];

            if (my_min_edge != -1)
            {
                int v = get_neighbour(my_min_edge);

                if (comp_min_edge == -1)
                {
                    d_comp_min_edge[my_comp] = my_min_edge;
                    *found_min_edge = false;
                }
                else
                {
                    int curr_min_neigh = get_neighbour(comp_min_edge);
                    int my_min_edge_weight = get_weight(my_min_edge), comp_min_edge_weight = get_weight(comp_min_edge);
                    if (my_min_edge_weight < comp_min_edge_weight || (my_min_edge_weight == comp_min_edge_weight && d_parent[v] < d_parent[curr_min_neigh]))
                    {
                        d_comp_min_edge[my_comp] = my_min_edge;
                        *found_min_edge = false;
                    }
                }
            }
        }
        end;
    }

    // remove cycles of 2 nodes
    forall(n_vertices, NUM_THREADS)
    {
        // if u is the representative of its component
        if (d_parent[u] == get_node(u))
        {
            int comp_min_edge = d_comp_min_edge[u];
            if (comp_min_edge != -1)
            {
                int v = get_neighbour(comp_min_edge);
                int parent_v = d_parent[v];

                int v_comp_min_edge = d_comp_min_edge[parent_v];
                if (v_comp_min_edge != -1)
                {
                    // v is comp(u)'s neighbour
                    // w is comp(v)'s neighbour
                    int w = get_neighbour(v_comp_min_edge);
                    if (d_parent[u] == d_parent[w] && d_parent[u] < d_parent[v])
                    {
                        d_comp_min_edge[d_parent[v]] = -1;
                    }
                }
            }
        }
    }
    end;

    // update the MST edges
    forall(n_vertices, NUM_THREADS)
    {
        // if u is the representative of its component
        if (d_parent[u] == get_node(u))
        {
            int curr_comp_min_edge = d_comp_min_edge[u];
            if (curr_comp_min_edge != -1)
            {
                d_edge_in_mst[curr_comp_min_edge] = 1;
                ATOMIC_INT atomic_data(*counter);
                atomic_data += 1;
            }
        }
    }
    end;

    logfile << "Num Edges: " << *counter << std::endl;

    // update parents
    forall(n_vertices, NUM_THREADS)
    {
        // if u is the representative of its component
        if (d_parent[u] == get_node(u))
        {
            int curr_comp_min_edge = d_comp_min_edge[u];
            if (curr_comp_min_edge != -1)
            {
                *rem_comp = false;
                int v = get_neighbour(curr_comp_min_edge);
                d_parent[u] = d_parent[v];
            }
        }
    }
    end;

    // flatten parents
    *parents_updated = false;
    while (!*parents_updated)
    {
        *parents_updated = true;
        forall(n_vertices, NUM_THREADS)
        {
            int parent_u = d_parent[u];
            int parent_parent_u = d_parent[parent_u];

            if (parent_u != parent_parent_u)
            {
                *parents_updated = false;
                d_parent[u] = parent_parent_u;
            }
        }
        end;
    }
}
```