**Algorithm 1:**

Procedure Shiloach-Vishkin (V, E)

1. for all v ∊ V: ∏ (v) 🡨 v
2. hooking 🡨 true
3. while hooking do
4. hooking 🡨 false
5. for all u ∊ V do in parallel
6. for all v ∊ N (u) do in parallel
7. if ( ∏(u) < ∏(v) and ∏(v) = ∏(∏(v)) ) then
8. ∏(∏(v)) 🡨 ∏(u)
9. hooking 🡨 true
10. end if
11. end for
12. end for
13. for all v ∊ V do in parallel
14. while ∏(∏(v)) ≠ ∏(v) do
15. ∏(v) 🡨 ∏(∏(v))
16. end while
17. end for
18. end while
19. return ∏

In our case of connected components, we are using edges instead of vertices as the primitive entity.

Each edge will receive a unique ID, just like how each vertex receives a unique ID in the connected components.

Line 11, equitruss, don’t worry about the super node ID for now, because, as each edge has an unique ID given in previous step, when building up super node, the merging of each edge into existing super node will compare the current super node ID (which is the ID of the edge with minimum edge ID) with the new edge’s ID and take the minimum of the two.

The main problem is to find and pack the edges in the same super node, for that purpose, we need to find the intersection of the adjacency list to find the edges that has triangle connectivity with the current edge.

In connected components in a regular graph, we needed to scan the edge for grouping the components together. Instead of edge, we need to scan 2 things, 1) trussness k and 2) triangle connectivity

The thought occurred to me how about numbering the edges from 1 to m just like we do the numbering for vertices for 1 to n. Then we can run the original Shiloach-Vishkin algorithm to find the connected components. In a regular graph, where we are finding connected components, we are basically using the edge connectivity (adjacency list) to run the hooking process. But in case of hooking for the edges

(1 …. m), do we have anything similar? There are 2 conditions that need to satisfy as mentioned above.

Let’s start with line no. 5, now instead of traversing over all the vertices u in parallel, we will be traversing over all the edges 1--> m in parallel.

In line number 6, instead of the adjacency list of a vertex u, we need to find all the edges that form triangle with edge (u, v). For that purpose, we need to have an intersection of N(u) and N(v) (here neighborhood of u is represented by N(u). Come to think of it, when I was computing the support of edge (u,v), does it make sense to store the common neighborhood list in an array during that time so that we do not need to recompute, it will be a compromise between storage efficiency for GPU as gpu has low storage and the computation cost, if we store the intersection list during support computation, we do not need to redo the same task again).

If we have the intersection, then we know we have to process edges up to the size of the intersection, if we assign each of those edges to parallel cuda thread, then we are stuck with one problem, how do we determine the thread number to assign each triangle, now we do know that each hooking operation will be handled by one cuda thread and if there is let’s say 5 edges found (does edge direction matter? Is it sufficient to process only one edge (u, w) or do we need to also process (w, u). For the connected components in the above algorithm, we needed to considered undirected graph, do we need to do the same for the connected components of edge, i.e., instead of 1 …. m edges, we need to consider 2m edges)

**Algorithm 2:**

Procedure Edge-Connected-Components-Parallel

1. for all e ∊ E: ∏ (e) 🡨 e
2. hooking 🡨 true
3. while hooking do
4. hooking 🡨 false
5. for all edge e1 (u, v) do in parallel
6. Intersect the adjacency list of e1 (u, v) to get all the edges making **k-triangle** with e1
7. for all the edges e2 making **k-triangle** with e1, do in parallel
8. if ( ∏(e1) < ∏(e2) and ∏(e2) = ∏(∏(e2)) and τ (e1) = τ (e2) ) then
9. ∏(∏(e2)) 🡨 ∏(e1)
10. hooking 🡨 true
11. end if
12. end for
13. end for
14. for all e ∊ E do in parallel
15. while ∏(∏(e)) ≠ ∏(e) do
16. ∏(e) 🡨 ∏(∏(e))
17. end while
18. end for
19. end while
20. return ∏

**Note:** The term k-triangle means all 3 edges in the triangle must have trussness >= k

A few items to figure out

* + - 1. How to uniformly number the super nodes (k-triangle-connected edges) in non-decreasing order (Algorithm 2, line 11).
      2. How to maintain a list of super node IDs for edge that has bigger trussness to the edge that has smaller trussness (Algorithm 2, line 31-33).
      3. How to create the super edges between super nodes (Algorithm 2, line 17 - 19)

Just gathering my thoughts here,

One way I can connect the super nodes by super edges is, while I am checking the k-triangle connectivity of the neighboring edges of an edge, I will see if the trussness of any neighboring edge is bigger than the trussness of the edge, if that’s the case, then I will keep a data structure of super edges, i.e., a vector of super edges with the edge (or index of the edge id). If I go in that direction, then I will have to wait until the end of Shiloach-Vishkin to get the final connected component IDs of the edges. This is because during the hooking-compression phase, it is not yet decided the final component ID (parent) of each edge. We will iterate over that vector where each element is a pair of edges, will connect two edges by creating a super edge by their connected component ID (parent). Now, a problem here is, what if there are multiple super edges essentially connecting the same super nodes, for instance, one super node SP1 connected to other super node SP2 by multiple super edges, if that is the case then we need to merge (remove duplicate) all those super edges. It brings another question, is it possible to actually have multiple super edges connecting the same super nodes, why or why not?

What we definitely know that one edge can connect multiple (more than two) super nodes. That is possible because consider the scenario where a triangle with all three different edges have different trussness. If that’s the case, all three of those edges will be part of three different connected components, and all of them will connect to/maintain 2 different super nodes (connected components).

**Algorithm 3:**

Procedure Parallel-Edge-Connected-Components- with-Super-Edge

1. for all e ∊ E: ∏ (e) 🡨 e
2. hooking 🡨 true
3. while hooking do
4. hooking 🡨 false
5. for all edge e1 (u, v) do in parallel
6. Intersect the adjacency list of e1 (u, v) to get all the edges making **k-triangle** with e1
7. for all the edges e2 making **k-triangle** with e1, do in parallel
8. **vector<pair<edge, edge>> sp\_edge**
9. if (∏(e1) < ∏(e2) and ∏(e2) = ∏(∏(e2)) and τ (e1) = τ (e2)) then
10. ∏(∏(e2)) 🡨 ∏(e1)
11. hooking 🡨 true
12. end if
13. **if (τ (e1) < τ (e2)) // upward super edge creation k < k1**
14. **sp\_edge.push\_back({e1, e2})**
15. **end if**
16. end for
17. end for
18. for all e ∊ E do in parallel
19. while ∏(∏(e)) ≠ ∏(e) do
20. ∏(e) 🡨 ∏(∏(e))
21. end while
22. end for
23. end while
24. **merge all thread local sp\_edge to get the summary graph**
25. return ∏

**Algorithm 4:**

k-iterative-Parallel-Connected-Components-with-super-edge

1. for all e ∊ E: ∏ (e) 🡨 e
2. map<pair<edge, edge>> sp\_edge //sp\_edge is thread local
3. Group set of edges into different subsets upon their trussness, k = 3, 4, ……., kmax
4. for k = kmin to kmax // 3 <= kmin <= kmax
5. hooking 🡨 true
6. while hooking do
7. hooking 🡨 false
8. for all edge e (u, v) ∊ Φ(k) do in parallel
9. Intersect the adjacency list of e(u, v) to get all the edges e1 (u, w) and e2 (v, w) making **k-triangle** with e
10. for all the edges e1, and e2 making **k-triangle** with e, do in parallel
11. if (∏(e) < ∏(e1) and ∏(e1) = ∏(∏(e1)) and τ (e) = τ (e1)) then
12. ∏(∏(e1)) 🡨 ∏(e)
13. hooking 🡨 true
14. end if
15. if (∏(e) < ∏(e2) and ∏(e2) = ∏(∏(e2)) and τ (e) = τ (e2)) then
16. ∏(∏(e2)) 🡨 ∏(e)
17. hooking 🡨 true
18. end if
19. k1 🡨 trussk[e1]
20. k2 🡨 trussk[e2]
21. lowest\_k 🡨 min (k, k1, k2)
22. if (k > lowest\_k and lowest\_k = k1) //downward super-edge creation k > k1
23. sp\_edge.insert({∏(e1), ∏(e)})
24. end if
25. if (k > lowest\_k and lowest\_k = k2) //downward super-edge creation k > k2
26. sp\_edge.insert({∏(e2), ∏(e)})
27. end if
28. end for
29. end for
30. for all e ∊ Φ(k) do in parallel
31. while (∏(∏(e)) ≠ ∏(e)) do
32. ∏(e) 🡨 ∏(∏(e))
33. end while
34. end for
35. end while
36. k 🡨 k+1
37. end for
38. merge all thread local sp\_edge to get the summary graph
39. return ∏

**Algorithm 5:**

k-iterative-Parallel-Connected-Components-with-super-edge-O(logd+1)-hooking-operations

1. for all e ∊ E: ∏ (e) 🡨 e
2. map<pair<edge, edge>> sp\_edge //sp\_edge is thread local
3. Group set of edges into different subsets upon their trussness, k = 3, 4, ……., kmax
4. for k = kmin to kmax // 3 <= kmin <= kmax
5. hooking 🡨 true
6. while hooking do
7. hooking 🡨 false
8. for all edge e (u, v) ∊ Φ(k) do in parallel
9. Intersect the adjacency list of e(u, v) to get all the edges e1 (u, w) and e2 (v, w) making **k-triangle** with e
10. for all the edges e1, and e2 making **k-triangle** with e, do in parallel
11. if (∏(e) < ∏(e1) and ∏(e1) = ∏(∏(e1)) and τ (e) = τ (e1)) then
12. ∏(∏(e1)) 🡨 ∏(e)
13. hooking 🡨 true
14. end if
15. if (∏(e) < ∏(e2) and ∏(e2) = ∏(∏(e2)) and τ (e) = τ (e2)) then
16. ∏(∏(e2)) 🡨 ∏(e)
17. hooking 🡨 true
18. end if
19. end for
20. end for
21. for all e ∊ Φ(k) do in parallel
22. while (∏(∏(e)) ≠ ∏(e)) do
23. ∏(e) 🡨 ∏(∏(e))
24. end while
25. end for
26. end while
27. Call ***procedure\_create\_super\_edge\_in\_parallel* (algorithm 5.5)**
28. k 🡨 k+1
29. end for
30. merge all thread local sp\_edge to get the summary graph
31. return ∏

**Algorithm 5.5:**

procedure\_create\_super\_edge\_in\_parallel

1. for all edge e (u, v) ∊ Φ(k) do in parallel
2. Intersect the adjacency list of e(u, v) to get all the edges e1 (u, w) and e2 (v, w) making **k-triangle** with e
3. for all the edges e1, and e2 making **k-triangle** with e, do in parallel
4. k1 🡨 trussk[e1]
5. k2 🡨 trussk[e2]
6. lowest\_k 🡨 min (k, k1, k2)
7. if (k > lowest\_k and lowest\_k = k1) //downward super-edge creation k > k1
8. sp\_edge.insert({∏(e1), ∏(e)})
9. end if
10. if (k > lowest\_k and lowest\_k = k2) //downward super-edge creation k > k2
11. sp\_edge.insert({∏(e2), ∏(e)})
12. end if
13. end for
14. end for

**Algorithm 6:**

1. for all e ∊ E: ∏ (e) 🡨 e
2. map<pair<edge, edge>> sp\_edge //sp\_edge is thread local
3. Group set of edges into different subsets upon their trussness, k = 3, 4, ……., kmax
4. for k = kmin to kmax // 3 <= kmin <= kmax
5. hooking 🡨 true
6. while hooking do
7. hooking 🡨 false
8. for all edge e (u, v) ∊ Φ(k) do in parallel
9. Intersect the adjacency list of e(u, v) to get all the edges e1 (u, w) and e2 (v, w) making **k-triangle** with e
10. for all the edges e1, and e2 making **k-triangle** with e, do in parallel
11. if (∏(e) < ∏(e1) and ∏(e1) = ∏(∏(e1)) and τ (e) = τ (e1)) then
12. ∏(∏(e1)) 🡨 ∏(e)
13. hooking 🡨 true
14. end if
15. if (∏(e) < ∏(e2) and ∏(e2) = ∏(∏(e2)) and τ (e) = τ (e2)) then
16. ∏(∏(e2)) 🡨 ∏(e)
17. hooking 🡨 true
18. end if
19. end for
20. end for
21. for all e ∊ Φ(k) do in parallel
22. while (∏(∏(e)) ≠ ∏(e)) do
23. ∏(e) 🡨 ∏(∏(e))
24. end while
25. end for
26. end while
27. k 🡨 k+1
28. end for
29. Call ***procedure\_create\_super\_edge\_in\_parallel* (algorithm 6.5)**
30. merge all thread local sp\_edge to get the summary graph
31. return ∏

**Algorithm 6.5:**

procedure\_create\_super\_edge\_in\_parallel

1. for all edge e (u, v) ∊ E do in parallel
2. Intersect the adjacency list of e(u, v) to get all the edges e1 (u, w) and e2 (v, w) making **k-triangle** with e
3. for all the edges e1, and e2 making **k-triangle** with e, do in parallel
4. k1 🡨 trussk[e1]
5. k2 🡨 trussk[e2]
6. lowest\_k 🡨 min (k, k1, k2)
7. if (k > lowest\_k and lowest\_k = k1) //downward super-edge creation k > k1
8. sp\_edge.insert({∏(e1), ∏(e)})
9. end if
10. if (k > lowest\_k and lowest\_k = k2) //downward super-edge creation k > k2
11. sp\_edge.insert({∏(e2), ∏(e)})
12. end if
13. end for
14. end for