Parallelizing Single Source Shortest Path using GPU

Dhruv Dogra, Rutgers University

Abstract

In this project we parallelize single source shortest path problem and analyze its performance on different graphs. We implement two different algorithms, namely Bellman Ford and work efficient, on GPU.

1. Introduction

In graph theory, a single source shortest path (SSSP) algorithm finds a path starting from a source vertex v to all other vertices in the graph. This problem has many applications like route planning, AI, 3D modeling, and social networks. Therefore it has been extensively been studied and many algorithms are there which can solve this problem. We take a look at two of the many solutions for this problem.

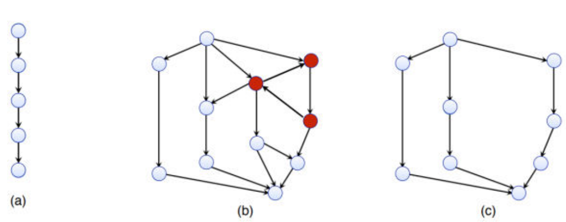
In SSSP there exists an inherent parallelism which we will exploit in our GPU implementation. Before discussing the algorithms let us look at some definitions involving graphs.

Vertex: A vertex or a node is a fundamental unit that makes up a graph.

Edge: An edge is set of vertices. It may or may not be ordered.

Graph: A graph G consists of a set of vertices V and a set of edges E, where an edge is an unordered pair of vertices.

DAG: DAG stands for Directed Acyclic Graph which is a finite directed graph with no directed cycles. That is, it consists of finitely many vertices and edges, with each edge directed from one vertex to another, such that there is no way to start at any vertex v and follow a consistently-directed sequence of edges that eventually loops back to v again.



In the figure shown above, graphs A and C are DAGs but graph B is not a DAG as it has a cycle.

Next let us look at the input to our algorithms. The input file is a plain text file that contains only edges. Each line in the input file corresponds to an edge in the graph and has at least two vertex indices separated by space or tab edge in the graph and has at least two vertex indices separated by space or tab. The first number specifies the start point of the edge of interest and the second one specifies the end point (the node that is pointed to in the edge). The third number (optional) contains the weight of the edge, if not specified, the edge weight is set to 1 by default.

We shall analyze our algorithm over the following input graphs:

1. RoadNetCA: http://snap.stanford.edu/data/roadNet-CA.htm
2. LiveJournal: http://snap.stanford.edu/data/soc-LiveJournal1.html
3. Pokec: http://snap.stanford.edu/data/soc-pokec.html
4. HiggsTwitter: http://snap.stanford.edu/data/higgs-twitter.html
5. WebGoogle: http://snap.stanford.edu/data/web-Google.html
6. Amazon0312: http://snap.stanford.edu/data/amazon0312.html

Now that our definitions are clear and inputs are defined we analyze our two implementations in the next sections.

2. Bellman Ford Algorithm

In this algorithm we compute shortest paths from a single source vertex to all of the other vertices in a weighted digraph.

Figure below illustrates the pseudo code for sequential Bellman Ford.

```
for each vertex v in vertices:
        distance[v] := inf;
distance[source] = 0;
for i from 1 to size(vertices)-1 {
  for each edge (u, v) with weight w in edges {
      if distance[u] + w < distance[v]:
        distance[v] := distance[u] + w;
        }
  if (no node's_distance_changed)_break;
}
```

We first calculate the shortest distances which have atmost one edge in the path. Then, we calculate shortest paths with at most 2 edges, and so on. After the i^{th} iteration of outer loop, the shortest paths with at most i edges are calculated. There can be maximum |V| – 1 edge in any simple path. That is why the outer loop runs |v| – 1 time. So if we calculate shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest path with at-most (i+1) edges.

This algorithm can be parallelized. We can parallel perform computation on the inner for loop. This loop processes edges in outer loop iteration. We can divide total edge to be processes evenly across all processors. The pseudo code for parallel Bellman Ford is shown below-

```
kernel edge_process(L, distance_prev, distance_cur)
{
  load = L.length % warp_num == 0 ? L.length/warp_num: L.length/warp_num+1;
  beg = load*warp_id;
  end = min(L.length, beg + load);
  beg = beg + lane_id;
  for ( i = beg, i < end,  i += 32 ) {
    u = L[i].src;
    v = L[i].dest;
    w = L[i].weight;
    if ( distance_prev[u] + w < distance_prev[v]
      atomicMin(&distance_cur[v], distance_prev[u] + w);

  }
}
...

for (i from 1 to size(vertices)-1 ) {
  edge_process(L, distance_prev, distance_cur);
  if ( no node is changed) break;
  else swap(distance_cur, distance_prev);
}
```

This is an out-of-core implementation which means that there are two different arrays which contain updated and original distances. We also analyze an incore implementation where there shall be a single array containing distance values. To get a deeper understanding we look at performance first with source vertex and then with destination vertex sorted. C++ provides a quick sort method which we have used to implement sorting. Using library method is better than creating your own sort method because former methods undergo much rigorous testing and are highly optimized compared to latter ones.

Now we look at the tables containing data of our implementation. We tried five different block size and block number configurations.

Table 1: Outcore; Sorted by Source

Graph	256 x 8	384 x 5	512 x 4	768 x 2	1024 x 2
Amazon0312	339.027	342.041	336.602	358.728	345.591
HiggsTwitter	10.192	10.567	10.29	11.896	10.241
LiveJournal	3482.59	3480.47	3511.14	3456.1	3659.6
Pokec	22.636	23.235	22.621	26.567	22.592
RoadNetCA	6671.53	6668.35	6646.36	6922.66	6604.39
WebGoogle	593.224	586.364	584.115	581.59	581.57

Table 2: Outcore; Sorted by Destination

Graph	256 x 8	384 x 5	512 x 4	768 x 2	1024 x 2
Amazon0312	491.513	487.40	486.471	463.042	484.927
HiggsTwitter	63.991	62.857	63.926	60.112	64.088
LiveJournal	4305.81	4302.4	4349.52	4216.78	4353.31
Pokec	7528.79	7476.7	7536.02	7507.74	7431.41
RoadNetCA	142.635	140.94	142.791	133.372	142.814
WebGoogle	961.997	962.02	960.601	926.774	955.054

Table 3: Incore; Sorted by Source					
Graph	**256 x 8**	**384 x 5**	**512 x 4**	**768 x 2**	**1024 x 2**
Amazon0312	124.904	128.763	124.2	117.295	123.177
HiggsTwitter	10.331	10.576	10.192	11.966	10.257
LiveJournal	1844.06	2115.83	1843.01	1799.59	1821.82
Pokec	142.84	140.966	142.746	133.327	143.128
RoadNetCA	3289.18	3294.39	3276.07	3360.98	3218.59
WebGoogle	312.282	313.105	351.907	330.391	331.233

Table 4: Incore; Sorted by Destination					
Graph	**256 x 8**	**384 x 5**	**512 x 4**	**768 x 2**	**1024 x 2**
Amazon0312	155.723	156.554	156.077	153.926	153.929
HiggsTwitter	64.018	62.845	63.954	60.076	64.138
LiveJournal	2117.39	2140.43	2138.34	2089.01	2145.67
Pokec	3797.27	3749.17	3761.58	3711.65	3750.3
RoadNetCA	142.876	140.938	142.724	133.318	142.84
WebGoogle	429.205	507.444	428.225	462.53	427.005

In this algorithm we also implemented a shared memory version for kernel process. We applied segment scan type method on the edges sorted by destination vertex. Only out-of-core method was used run this shared memory method.

Table 5: Outcore; Shared Memory; Sorted by Destination					
Graph	**256 x 8**	**384 x 5**	**512 x 4**	**768 x 2**	**1024 x 2**
Amazon0312	697.705	583.30	579.842	590.083	568.384
HiggsTwitter	85.132	75.682	74.139	73.054	73.338
LiveJournal	5875.63	4896.2	4888.98	4844.2	4897.65
Pokec	13379.4	10324.	10223.4	10615.9	10173.8
RoadNetCA	181.086	163.45	160.164	157.958	158.114
WebGoogle	1127.42	1035.2	1025.76	1002.71	1013.62

2.1 Observations

1. Looking at performance time we can see that edges sorted by destination perform worse as compared to edges sorted by sources. This can be attributed to lesser thread collisions. Fewer threads try to update same vertex when source based sorting is done.
2. Shared memory method performed worse as compared to normal implementation. It could be due to the deficiencies present in segment scan approach. Segment scan suffers from thread divergence and memory coalescing. It could increase time to update vertex value.
3. As the block size was increased performance improved for all different algorithms. Data shows that best performance was obtained for 768 x 2 configuration and 1024 x 2 made no improvement.

3. Work Efficient Algorithm

In this algorithm reduce the number of edges processed in each iteration thereby improving overall performance. Based on the Bellman Ford description we can say that, if the starting point of the edge did not change, it will not affect the point-to end of the edge (the destination node on the other end). Thus, we can skip this edge when performing the comparison and updating computation.

We filter out the edges whose starting point did not change in the last iteration and only keep the edges that might lead to a node distance update. We use parallel exclusive scan operation to remove unwanted edges. In the source code we used exclusive scan method provided by CUDA Thrust library. Additionally we employed warp voting algorithm like *mask* and *ballot* to get edges with updated sources.

Following tables contains data we obtained after performing incore and outcore methods. Besides kernel computational we also calculated total time to filter out edges.

OUT-OF-CORE; Sorted by SOURCE

Table 6.1: Computational Time

Graph	256 x 8	384 x 5	512 x 4	768 x 2	1024 x 2
Amazon0312	287.328	289.69	278.49	299.091	282.254
HiggsTwitter	8.969	8.331	8.93	9.68	8.864
LiveJournal	3407.36	3401.5	3464.97	3436.01	3439.76
Pokec	17.926	18.041	17.914	20.311	19.936
RoadNetCA	6620.2	6641.4	6632.11	6871.77	6640.75
WebGoogle	568.502	577.87	569.45	574.34	564.6

Table 6.2: Filter Time

Graph	256 x 8	384 x 5	512 x 4	768 x 2	1024 x 2
Amazon0312	268.704	277.91	270.86	299.957	270.152
HiggsTwitter	0	0	0	0	0
LiveJournal	1998.84	2024.14	1990.1	2223.07	1988.03
Pokec	0	0	0	0	0
RoadNetCA	6557.59	6625.67	6517.4	7314.44	6535.58
WebGoogle	336.665	336.872	333.21	366.138	331.178

OUT-OF-CORE; Sorted by DESTINATION

Table 7.1: Computational Time

Graph	256 x 8	384 x 5	512 x 4	768 x 2	1024 x 2
Amazon0312	443.475	453.057	461.67	463.515	475.696
HiggsTwitter	57.33	56.718	57.626	54.38	57.239
LiveJournal	4229.41	4229.81	4236.0	4214.62	4257.49
Pokec	7494.75	7437.91	7471.3	7570.98	7441.88
RoadNetCA	120.17	128.472	130.01	132.009	139.838
WebGoogle	944.4	947.17	942.5	947.54	940.04

Table 7.2: Filter Time					
Graph	**256 x 8**	**384 x 5**	**512 x 4**	**768 x 2**	**1024 x 2**
Amazon0312	822.447	826.989	820.52	808.227	817.551
HiggsTwitter	0	0	0	0	0
LiveJournal	7187.32	7122.64	7243.4	7013.35	7252.48
Pokec	0	0	0	0	0
RoadNetCA	10786	10724.8	10901	10525.9	10605.2
WebGoogle	1701.06	1682.65	1697.8	1624.13	1701.68

INCORE; SORTED BY SOURCE

Table 8.1: Computational Time					
Graph	**256 x 8**	**384 x 5**	**512 x 4**	**768 x 2**	**1024 x 2**
Amazon0312	116.41	118.798	120.04	115.049	120.461
HiggsTwitter	9.896	9.233	8.798	9.558	9.816
LiveJournal	1810	2090.46	1809.4	1753.18	1790.26
Pokec	32.926	33.99	32.924	39.276	32.879
RoadNetCA	4117.3	4119.01	4099.7	4099.74	4063.55
WebGoogle	380.40	362.967	378.35	350.296	378.57

Table 8.2: Filter Time					
Graph	**256 x 8**	**384 x 5**	**512 x 4**	**768 x 2**	**1024 x 2**
Amazon0312	119.3	121.93	120.951	134.27	121.529
HiggsTwitter	0	0	0	0	0
LiveJournal	1010.2	1022.2	1009.65	1108.1	1011.27
Pokec	0	0	0	0	0
RoadNetCA	3535.8	3533.7	3515.18	3697.2	3524.56
WebGoogle	159.25	158.97	155.66	161.44	157.313

INCORE; SORTED BY DESTINATION

Table 9.1: Computational Time					
Graph	**256 x 8**	**384 x 5**	**512 x 4**	**768 x 2**	**1024 x 2**
Amazon0312	113.43	128.241	125.971	131.213	140.488
HiggsTwitter	57.405	56.815	57.542	54.373	57.22
LiveJournal	2082.9	2120.03	2116.3	2648.68	2947.26
Pokec	3745.0	3741.04	3713.37	3505.77	3724.97
RoadNetCA	150.19	148.528	150.053	142.079	149.811
WebGoogle	559.19	524.775	592.527	521.404	556.77

Table 9.2: Filter Time					
Graph	**256 x 8**	**384 x 5**	**512 x 4**	**768 x 2**	**1024 x 2**
Amazon0312	302.6	323.93	321.588	298.806	299.771
HiggsTwitter	0	0	0	0	0
LiveJournal	4123.2	3580.9	4157.94	3518.81	4154.1
Pokec	0	0	0	0	0
RoadNetCA	5801.9	5644.8	5824.68	5240.31	5712.42
WebGoogle	853.10	793.39	903.328	762.706	849.087

3.1 Observations

1. The computational time for this implementation is better than previous one. But the time taken to perform filtering is high. Most of the total time is spent on the filtering step. Additionally if the graph is size large then filtering step takes a lot of time.
2. Edges sorted by destinations perform worse than ones sorted by source. This is due to threads trying to update same content which in turns serializes the process.
3. The total iteration of this implementation should be lesser than previous one as this implementation has a preprocessing step involved.

4. References

- Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

- A. Davidson, S. Baxter, M. Garland, and J. D. Owens. Work-efficient parallel gpu methods for single-source shortest paths. In 2014 IEEE 28th International Parallel and Distributed Processing Symposium, pages 349–359, May 2014

- https://www.udacity.com/course/intro-to-parallel-programming--cs344

- Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.