Optimizing Single Source Shortest Path on GPU

Dhruv Dogra, Rutgers University

Abstract

In this project we optimize single source shortest path algorithm implemented in the previous project. We achieve this by modifying the filtering stage of the algorithm. Then we compare our current implementation against the work efficient algorithm and make observations.

1. Introduction

Lets refresh our previous implementation known as work efficient algorithm. In that algorithm we reduce the number of edges processed in each iteration. This in turns improves overall performance. It is based on the assumption that if the starting point of the edge does not change then the destination won't be affected.

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.

Though this approach is better than the naive implementation there is still room for improvements. We propose an algorithm reduces the overhead present in the filtering stage. This approach uses the adjacency list of all the changed vertices to calculate the shortest path. This algorithm can be divided into major parts which are mentioned below:

1. Indexing outgoing edges of changed nodes

2. Generating to-process edge list

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

In the next two sections we'll look at the two main parts of our optimized algorithm.

2. Indexing Outgoing Edges of Changed Nodes

Before looking in depth explanation we would like to clarify that we have only implemented an out-of-core version with nodes sorted by source vertex. Before executing this step we need to create few data structures which are shown below:

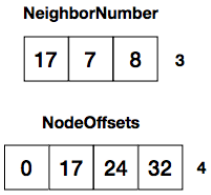
AllNodes, as the name suggests, is the collection of all the nodes in the graph. *AllNeighborNumber* contains the number of outgoing edges for a vertex. *AllOffsets* is obtained by performing exclusive scan on the AllNeighborNumber array. *AllOffsets[i]* gives the start index of vertex i's adjacency list in *Edges* (it is the array containing edges and it's sorted by source vertex). Using this information we can compute the list of to-process edges in parallel.

Let us look at the psuedocode the generates the *NodeQueue* which contains changed/updated vertices.

Listing 1: Generate NodeQueue

```
1
2  if (distancePrev[u] + w < distancePrev[v]
3          && distancePrev[u] + w < distanceCur[v]) {
4          // the atomic min ensures that only one thread adds v to NodeQueue
5          old_val = atomicMin(&distanceCur[v], distancePrev[u] + w);
6
7          if (old_val >= distancePrev[v] && distancePrev[u] + w < old_val) {
8                  int idx = atomicAdd(&QueueCounter, 1);
9                  NodeQueue[idx] = v;
10         }
11 }
```

Using the NodeQueue array we generate two more data structures.



NeighborNumber is an array containing number of neighbors of every vertex present in *NodeQueue*. We get *NodeOffsets* by performing exclusive prefix scan on NeighborNumber array. *NodeOffsets* contains information about the offsets at which each vertex's outgoing edges will appear in to-process edge list.

After the kernel call we swap the distancePrev and distanceCurr arrays and check the size of NodeQueue. If the NodeQueue is empty we break out of the loop.

3. Generating the To-Process Edge List

Now that we have found the vertices that have changed we use this information to get edges that need to be processed. The NodeOffset array contains the total edges that need to be processed. We first divide the

Listing 2: Computing Thread Edge Ranges

```
1  if (total_edges % num_threads == 0) {
2          loadPerThread = NodeOffsets[QueueNodeSize] / num_threads
3  } else {
4          loadPerThread = NodeOffsets[QueueNodeSize] / num_threads + 1
5  }
6  tid.begin = i * loadPerThread
7  tid.end = min(((i + 1) * loadPerThread) - 1, NodeOffsets[QueueNodeSize] - 1)
```

workload evenly among threads and then update the to-process edge (tpe) list.

Having assigned a range we then make a thread do binary search its begin value on *NodeOffset* array, returning index with the closest value less than or equal to it. This index also gives the outgoing edges we need to add in tpe.

Listing 3: Accumulating To Process Edges

```
1  i = binarySearch(NodeOffsets, tid.begin);
2  startVertex = QueueNode[i];
3  phi = x - NodeOffsets[i];
4  // i is the current index in the NodeOffsets/NodeQueue arrays
5  for x in [tid.begin, tid.end] {
6      if (x < NodeOffsets[i+1]) {
7          e = AllOffsets[startVertex] + phi;
8          ToProcessEdges[x] = e;
9      } else { // we need to read from a new source vertex
10             i++;
11             if (i >= NodeQueueSize) {
12                     cerr << "Error, NodeQueue access out of bounds\n";
13                     exit(EXIT_FAILURE);
14             }
15             startVertex = QueueNode[i];
16             phi = x - NodeOffsets[i];
17             e = AllOffsets[startVertex] + phi;
18             ToProcessEdges[x] = e;
19     }
20     phi++;
21 }
```

The filtering algorithm is shown the Listing 3.

An if-else is present to handle the case when a thread needs to add edges that spread across offset array.

Next we look at the data collected after performing work efficient and optimized implementation. The algorithms were run on GPU GT 730 with 2GB Memory, 384 CUDA cores, CUDA capability 3.5.

Table 1: Work Efficient; Compute Time

Graph	256 x 8	384 x 5	512 x 4	768 x 2	1024 x 2
Amazon0312	225.977	244.069	229.883	280.8	226.661
HiggsTwitter	10.907	11.034	10.645	13.757	10.789
LiveJournal	2228.03	2699.87	2230.83	2334.12	2312.78
Pokec	23.635	24.862	24.011	29.102	23.9
RoadNetCA	4352.07	4612.31	4360.93	5044.89	4362.97
WebGoogle	422.434	448.27	423.509	478.555	421.457

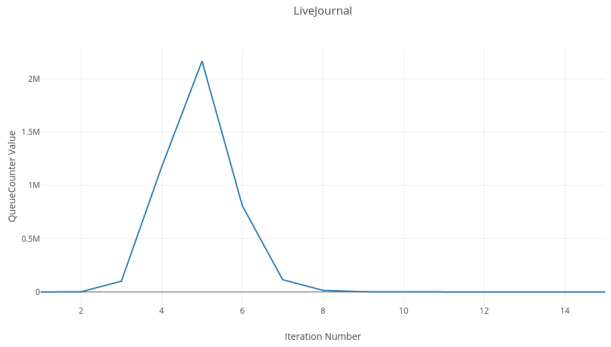
Table 2: Optimized; Compute Time

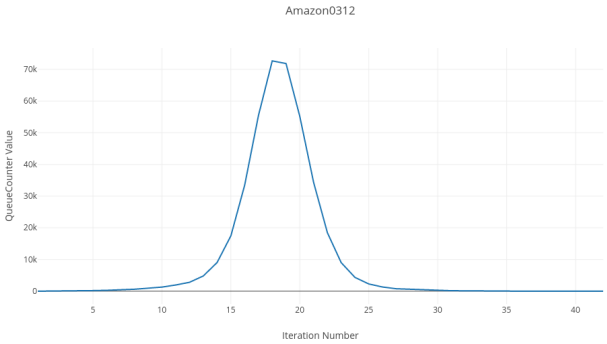
Graph	256 x 8	384 x 5	512 x 4	768 x 2	1024 x 2
Amazon0312	177.969	192.464	177.997	204.054	179.421
HiggsTwitter	14.13	15.032	14.111	15.901	14.097
LiveJournal	1273.63	1357.49	1268.29	1527.08	1304.64
Pokec	29.503	32.344	29.381	32.326	29.719
RoadNetCA	4658.88	4923.86	4675.68	4957.71	4684.59
WebGoogle	240.718	207.281	236.008	276.192	236.562

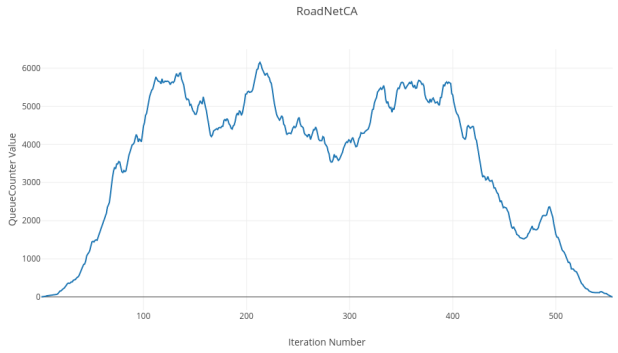
Table 3: Work Efficient; Filter Time					
Graph	256 x 8	384 x 5	512 x 4	768 x 2	1024 x 2
Amazon0312	313.452	317.358	312.998	353.212	303.012
HiggsTwitter	0	0	0	0	0
LiveJournal	2371.16	2675.58	2364.88	2575.2	2302.62
Pokec	0	0	0	0	0
RoadNetCA	5722.14	5791.65	5730.69	5181.23	5640.18
WebGoogle	389.162	388.062	388.954	437.974	377.956

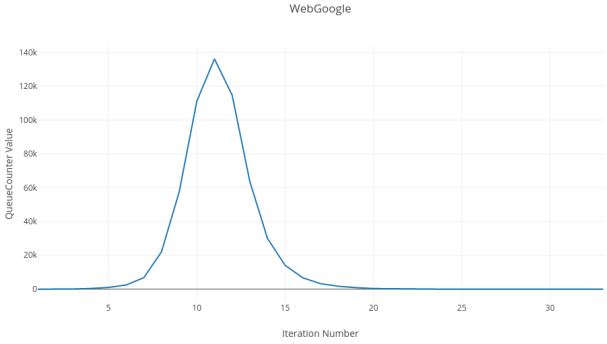
Table 4: Optimized; Filter Time					
Graph	256 x 8	384 x 5	512 x 4	768 x 2	1024 x 2
Amazon0312	288.561	284.008	292.636	278.553	292.4
HiggsTwitter	0	0	0	0	0
LiveJournal	3444.66	3223.41	3438.13	3367.32	3419.13
Pokec	0	0	0	0	0
RoadNetCA	5139.03	5161.78	5153.82	5169.42	5162.28
WebGoogle	359.571	288.087	352.076	343.173	351.992

Besides checking the running time we also looked at variance in the size of *NodeQueue*. The size of this data structure decides if the program will execute further or not. Following are the line charts depicting the change in value of *NodeQueue*. The x-axis indicates the number of iteration and the y-axis represents size of our data structure.









3. Observations

1. Looking at compute time we can see that optimized algorithm performs a lot better than work efficient one. The speedup of 2X can be observed. This can be attributed to fewer edges that are processed. Also fewer threads leads to lesser thread collision thereby improving time.
2. Filter time for work efficient and optimized looks comparable. This is due to time consuming memory copying operations occurring for the optimized algorithm. In each iteration we reinitialize the tpe array before making the second kernel call. After the kernel call we also copy new the from device to host which further consumes time. Though not much gain is observed in filter time we have managed to reduce compute time significantly.
3. The size of NodeQueue starts small, reaches maximum value and then drops to zero. This is expected as the number of updated/changed vertices increases their corresponding neighbors will also increase. This leads to a maximum value which then drops to zero as all the vertices get processed.

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.