Single Source Shortest Path on GPU

Ashish Jindal, Rutgers University

Abstract

This report presents analysis of some implementations for single source shortest path algorithm on GPU. First implementation tries to make bellman ford's sequential algorithm, parallel and analyse runtime in various scenarios considering order of edges in the input and use of shared memory and segmented scan to improve efficiency. Second implementation is a work efficient variation of the first implementation which tries to decrease the workload at each iteration in bellman ford algorithm.

1. List of graphs used for generating stats

RoadNetCA:

http://snap.stanford.edu/data/roadNet-CA.html

LiveJournal:

http://snap.stanford.edu/data/soc-LiveJournal1.html

WebGoogle:

http://snap.stanford.edu/data/web-Google.html

Amazon0312:
http://snap.stanford.edu/data/amazon0312.html

msdoor:
http://www.cise.ufl.edu/research/sparse/matrices/INPRO/msdoor.html

road-cal:
http://www.dis.uniroma1.it/challenge9/download.shtml
(CAL - California and Nevada - Distance Graph)

2. Implementation using Bellman Ford

The sequential bellman-ford algorithm checks all edges at every iteration and updates a node if the current estimate of its distance from the source node can be reduced. The number of iterations is at most the same as the number of vertices if no negative cycle exists. The complexity of the sequential bellman-ford algorithm is O(| V | · | E |).

In the parallel bellman-ford algorithm, we exploit the parallelism of edge processing at every iteration. Each of the input edges is checked at every iteration, we can distribute these edges evenly to different processors such that each processor is responsible for the same number of edges.

Following are the results for various configurations **1024x2 -**

Graph	incore src	incore dest	outcore src	outcore dest	outcore shared mem
amazon 0312	156.2	164.8	439.3	463.9	1055.9
msdoor	2600.6	2609.5	6904.2	6874	27947
roadnet-ca	4391.1	4386.3	7011.4	7376.6	22461.9
live journal	1931.9	2224.4	4183.8	4203.2	9086.5
road-cal	28683.5	28369.5	48837.4	47809.5	144212
web-google	510.8	470.6	894.1	922.7	1736.6

Following are the **number of iterations** to complete the algorithm for various configurations **1024x2 -**

Graph	incore src	incore dest	outcore src	outcore dest	outcore shared mem
amazon 0312	15	15	42	40	42
msdoor	165	168	439	439	439
roadnet-ca	363	363	555	555	555
live journal	7	8	15	15	15
road-cal	2620	2630	4165	4165	4165
web-google	19	18	33	33	33

Following are the results from varying block size -

Incore - Src sorted -

Graph	256x8	384x5	512x4	768x2	1024x2
amazon 0312	151.3	160.4	157.3	145.2	156.2
msdoor	2669.5	2727.9	2687.9	2638.8	2600.6
roadnet-ca	4444.1	4511.5	4433.5	4302.1	4391.1
live journal	1940.7	2223.6	1932.4	1903.4	1931.9
road-cal	29232.9	30188.8	29022.9	28841.2	28683.5
web-google	515.3	486.4	541.1	505.7	510.8

Following are the results from varying block size -

Incore - Dest sorted

Graph	256x8	384x5	512x4	768x2	1024x2
amazon 0312	164.8	167.8	165.2	163.7	164.8
msdoor	2723.6	2707.4	2684.6	2670.1	2609.5
roadnet-ca	4421.7	4497.9	4411.7	4341.6	4386.3
live journal	2188.9	2216.4	2210.8	2183.6	2224.4
road-cal	28869.7	30043.2	28622.8	28327.7	28369.5
web-google	444.9	527.4	500.5	495.2	470.6

Following are the results from varying block size -

Outcore - Src sorted

Graph	256x8	384x5	512x4	768x2	1024x2
amazon 0312	444.4	450.9	443.4	438.3	439.3
msdoor	7096.2	7352.3	7050.4	6944.9	6904.2
roadnet-ca	7196.1	7291.4	7167.5	6970.1	7011.4
live journal	4182.9	4153.1	4201.3	4106.5	4183.8
road-cal	50145.2	50741.2	49461.9	48852.2	48837.4
web-google	902.8	899.4	899.8	884.7	894.1

Following are the results from varying block size -

Outcore - Dest sorted

Graph	256x8	384x5	512x4	768x2	1024x2
amazon 0312	465.9	466.7	468.9	462.5	463.9
msdoor	7063.5	7297.1	7001.3	6942.2	6874
roadnet-ca	7151.1	7252.8	7111.1	6960.7	7376.6
live journal	4132.1	4143.7	4170.2	4121.6	4203.2
road-cal	48523.2	49566.9	48217.1	47770	47809.5
web-google	926.2	925.3	926.4	916.6	922.7

Following are the results from varying block size -

Outcore with shared mem - dest sorted

Graph	256x8	384x5	512x4	768x2	1024x2
amazon 0312	1225.6	968.2	981.4	1115.1	1055.9
msdoor	31563.7	27006.9	25056.2	29515.3	27947
roadnet-ca	25177.3	21713.9	21465.8	23482.7	22461.9
live journal	9721.7	8331.6	8501.4	9314.6	9086.5
road-cal	160607	136738	137129	149596	144212
web-google	1567.6	1560.8	1597.8	1772.6	1736.6

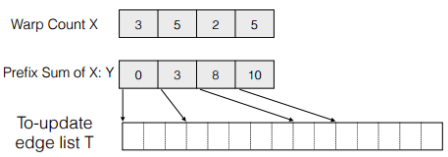
2.1. Analyses and Observations

1. Number of iterations for incore implementation is always less than that of the outcore version, probably because the parallel threads receive the updated vertex distances as soon as they are updated rather than wait for the next iteration.
2. When the input edges are sorted by destination node, the performance is worse compared to when they are sorted by source node in all most all cases. This can be because of many threads trying to update the same destination when edges are sorted in destination order. If they are sorted by source node there is a greater chance that parallel threads will be updating different destination vertices and thus will have less contention while writing data.
3. Shared memory version is performing really bad, I think this can be because of thread divergence in the segmentation step. Probably the cost of thread divergence here is much more than what we were trying to save while avoiding atomic operations.
4. For varying block size and block count the performance is not that much, but it can be observed that performance in case of 768 x 2 and 1024 x 2 is almost same in all cases, which is a bit surprising as the later one has more threads.

3. Work Efficient implementation

We can improve Implementation 1 by reducing the edges that need to be processed at each iteration. For any directed edge, if the starting point of the edge (the did not change, it will not affect the pointed-to end of the edge (the destination node on the other end). Thus, we can skip this edge when performing the comparison and updating "if (distance[u] + w < distance[v]) ..." computation.

Following image describes the steps involved in the filtering process -



In our implementation we have measured both processing time and filtering time and the stats are reported below –

Following are the **total-time** from various configurations **1024 x 2 -**

Graph	incore src	incore dest	outcore src	outcore dest
amazon 0312	322.9	768.3	329.2	867.8
msdoor	8722.6	10633.4	9761.2	11582.8
roadnet-ca	7525.9	11472.4	7878.4	11838.3
live journal	2022.2	6018	2780.7	7989.7
road-cal	52469.2	83708.4	59966.2	89994.1
web-google	421.4	1622.2	436.9	1789.3

Following are the **processing times** from various configurations **1024 x 2 -**

Graph	incore src	incore dest	outcore src	outcore dest
amazon0312	44.7	50.7	52.1	50.2
msdoor	699.8	1018.9	1304.2	1467.8
roadnet-ca	763.2	771.2	1176.4	1184
live journal	694.9	826.1	725.8	745.9
road-cal	8721.6	8739.4	16135.2	14903.1
web-google	86.2	84.9	92.6	93.1

Following are the **number of iterations** to complete the algorithm from various configurations **1024x2** -

Graph	incore src	incore dest	outcore src	outcore dest
amazon0312	42	37	42	41
msdoor	417	418	439	439
roadnet-ca	555	555	555	555
live journal	11	11	15	15
road-cal	4176	4159	4165	4165
web-google	32	33	33	33

Following are only the **processing times**, for filtering times you can refer the corresponding. stats file generated by running the provided script (impl1.sh).

Results from varying block size -

Incore - Src sorted

Graph	256x8	384x5	512x4	768x2	1024x2
amazon 0312	45.3	36.7	1064.1	35.7	44.7
msdoor	729.1	725.5	738.2	751.3	699.8
roadnet-ca	755.8	766.2	806.3	806.2	763.2
live journal	690.9	699.6	699.5	683.9	694.9
road-cal	8722.1	8798.5	8792.2	8642.4	8721.6
web-google	77.8	78.8	85.8	76.7	86.2

Following are the results from varying block size -

Incore - Dest sorted

Graph	256x8	384x5	512x4	768x2	1024x2
amazon 0312	148.9	167.2	49.2	50.9	50.7
msdoor	989.1	1079.8	1011.9	1030.9	1018.9
roadnet-ca	987.9	771.1	773.2	772.7	771.2
live journal	822.1	844.4	835.2	811	826.1
road-cal	8743.3	8657.3	8734.5	8637.8	8739.4
web-google	88.2	91.2	86.6	83.6	84.9

Following are the results from varying block size -

Outcore - Src sorted

Graph	256x8	384x5	512x4	768x2	1024x2
amazon 0312	52.3	44.1	43.7	52	52.1
msdoor	1307.4	1335.3	1316.5	1343.5	1304.2
roadnet-ca	1177.4	1177.1	1214.9	1177.7	1176.4
live journal	724.1	735.5	730.2	713.9	725.8
road-cal	16269.2	16393.1	16312.1	16095.2	16135.2
web-google	85.8	88.9	92.3	91.1	92.6

Following are the results from varying block size -

Outcore - Dest sorted

Graph	256x8	384x5	512x4	768x2	1024x2
amazon 0312	59.3	50.8	56	511.1	50.2
msdoor	1479.2	1503.6	1484.1	1504.3	1467.8
roadnet-ca	1417.2	1359.8	1184.7	1184.3	1184.0
live journal	740.5	751	747.3	734.1	745.9
road-cal	15068.8	15165.9	15068.1	14959.9	14903.1
web-google	97.8	96.7	95.6	94.3	93.1

3.1. Analyses and Observations

1. If we just take the processing times into account then they are way less than in the previous implementation, most of the time is spent in filtering step.
2. Very large amount of time goes in the filtering for some big graphs like live-journal and road-cal.
3. Number of iterations for incore implementation are much more than that in previous implementation - in some case twice. The iteration count is close to outcore version in previous implementation.
4. Number of iterations for incore implementation is always approximately same to that of outcore version.
5. When the input edges are sorted by destination node, the performance is worse compared to when they are sorted by source node in all most all cases. This can be because of many threads trying to update the same destination when edges are sorted in destination order. If they are sorted by source node there is a greater chance that parallel threads will be updating different destination vertices and thus will have less contention while writing data.

4. Implementation using Priority Queue

Not Working!

3. References

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

[2] 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.

[3] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for gpu computing. In Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware, GH '07, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[4] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nico, and K. Crowley. Principles of runtime support for parallel processors. In Proceedings of the 2Nd International Conference on Supercomputing, ICS '88, pages 140–152, New York, NY, USA, 1988. ACM.

[5] http://www.drdobbs.com/parallel/atomic-operations-and-low-wait-algorithm/240160177.

[6] https://docs.nvidia.com/cuda/cuda-c-programming-guide/.