Project 2: Parallelizing Single Source Shortest Path (SSSP) on GPU

Arthur Quintanilla, Rutgers University, New Brunswick, NJ
Ankith Kolisetti, Rutgers University, New Brunswick, NJ

INTRODUCTION:

This project report discusses the parallelizing single source shortest path (SSSP) algorithm on Graphical Processing Units using CUDA programming. Bellman-Ford Algorithm, with its massive inherent parallelism is a wide choice for implementing the SSSP method on parallel architecture.For this project, two versions of parallel SSSP implementations are done, which are both based on the Bellman-ford algorithm. The first version is the Bellman-ford algorithm, with its inherent parallelism allows us to exploit the parallelism of edge processing at every iteration. The second version is the Work Efficient algorithm, which is an improvement on Bellman-ford algorithm. Here, the implementation I is improved by reducing the edges that need to be processed at each iteration. To implement this, the parallel prefix sum operation is used for gathering to-process edges. After filtering out the edges that need not be considered, then the to-process edges are evenly distributed among different thread warps and the same iterative process is performed as in Implementation I.

1. Background

Single Source Shortest Path (SSSP) implementation has become important in the scientific, computer, and real world applications for finding the shortest path from from a single source node to all connected nodes [1]. The SSSP algorithms are most widely used in computer network routing, VLSI design, Artificial Intelligence and robotics, transport network, social networking, google maps, and 3D modeling. The data in these real time applications are represented in the form of graphs, which consist of millions of nodes and edges. The parallel SSSP algorithms are implemented on the GPUs using CUDA parallel programming.

A graph is a collection of nodes (vertices) and the links between these nodes are called the edges [7]. Each edge is related to an attribute, which is called the weight of the corresponding edge [7]. If each edge of a graph has a fixed starting point - source, and a destination node, then such a graph is called a directed graph [7]. Since the real time graphs have huge amount of data, and having millions and billions of vertices and edges, the problem solving becomes complex and time consuming. Hence, parallel SSSP algorithms are used to compute the shortest paths from the single source node to all other nodes in a graph [7]. The input to SSSP is a source node in a graph with v vertices and e directed edges, pointing in both directions [1]. This algorithm finds a path between two nodes of a weighted directed graph such that the sum of the weights of the edges creating this path is minimal [7].

The Graphics Processing Units (GPUs) provide us with a highly parallel, multi-threaded, and cost-efficient, many-core processing with tremendous computational power and a very high memory bandwidth [6]. Hence, the GPUs can be efficiently used to achieve substantially higher performance on intrinsically parallel computations. Many parallel programming languages such as NVIDIA's CUDA and OpenCL can be efficiently used on the GPUs for higher computing performance.

The Compute Unified Device Architecture (CUDA) is a parallel programming interface developed by NVIDIA that can be used to take advantage of the highly parallel architecture of the GPUs. CUDA is an extension of C programming language and works on Windows, Mac OS, and Linux distributions, and uses the CPU and GPU simultaneously for the execution of code [7].

2. Parallel Implementation

2.1.Implementation I: Bellman-ford Algorithm:

Bellman-Ford Algorithm is a SSSP finding algorithm that calculates shortest paths from a

single source vertex to all of the other vertices in a weighted directed graph [2].
Among different implementations of parallel algorithms for SSSP, Bellman-Ford algorithm is widely used due to its inherent massive parallelism. Bellman-Ford algorithm operates on all vertices independently, while each vertex maintains its distance to the source [1].

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, the parallelism of edge processing is exploited at every iteration. The efficiency of the parallel algorithm depends largely on the way the data is partitioned [6]. Each of the edges is checked at every iteration, hence the edges can be distributed evenly to different processors, so that each processor is responsible for the same number of edges.

2.2 Implementation II: Work Efficient Algorithm:

This implementation of Work Efficient algorithm is an improvement on Implementation I. Although, Bellman-Ford routing algorithm consists of massive inherent parallelism, it allows to consider all nodes in parallel and hence is not work efficient. Therefore, we implement a Work-Efficient algorithm, in which the edges that need to be processed at each iteration can be reduced by filtering out the edges whose starting point did not change in the last iteration, and hence keep only those edges that might lead to a node distance update. Since, in any directed edge, if the starting point of the edge did not change, then it will not affect the destination node on the other end, i.e, the pointed-to end of the edge.

Therefore, such edges can be skipped when performing the comparison and updating computations. The edges that need to be checked are referred to as the to-process edges. We can use parallel prefix sum operation to check the to-process edges. After filtering out the edges that need not be considered, and after gathering the to-process edges, the to-process edges are evenly distributed among different thread warps and the same iterative process as in Implementation I is performed. The focus here is on improving the performance of the computation stage and not the performance of the filtering stage.

3. REPORT RESULTS ON IMPLEMENTATION I:

3.1 Bellman-Ford Algorithm:

On graph p2p-Gnutella04.txt from http://snap.stanford.edu/data/p2p-Gnutella04.html .

Graph files that our program will be tested on were taking a lot of time to sort by destination or source. We decided to use much smaller graph files from the same web source provided in the assignment description.

Custom sorter executable was made (sorter.c) to make the default text files sorted according to source or destination nodes and were named Nutella04Source.txt and Nutella04Dest.txt in this example. To understand what was tested view test in makefile.

p2p-Gnutella04.txt file:

Configuration and time taken and Incore:

1. (256, 8) took 1.488ms
2. (384, 5) took 1.501ms
3. (512, 4) took 1.506ms
4. (768, 2) took 1.52ms
5. (1024, 2) took 1.498ms

p2p-Gnutella04.txt file:
Configuration and time taken and Out-of-core:

6. (256, 8) took 3.626ms
7. (384, 5) took 3.578ms
8. (512, 4) took 3.626ms
9. (768, 2) took 3.651ms
10. (1024, 2) took 3.607ms

Default file is not sorted by destination nodes, therefore we do not perform segment scan on it.

Nutella04Source.txt file (Sorted by source nodes):
Configuration and time taken and Incore:

11. (256, 8) took 1.303ms
12. (384, 5) took 1.321ms
13. (512, 4) took 1.316ms
14. (768, 2) took 1.333ms
15. (1024, 2) took 1.305ms

Nutella04Source.txt file:

Configuration and time taken and Out-of-core:

16. (256, 8) took 3.768ms
17. (384, 5) took 3.618ms
18. (512, 4) took 3.64ms
19. (768, 2) took 3.643ms
20. (1024, 2) took 3.618ms

Source sorted file is not sorted by destination nodes, therefore we do not perform segment scan on it

Nutella04Dest.txt file (Sorted by destination nodes):
Configuration and time taken and Incore:

21. (256, 8) took 1.598ms
22. (384, 5) took 1.59ms
23. (512, 4) took 1.662ms
24. (768, 2) took 1.579ms
25. (1024, 2) took 1.571ms

Nutella04Dest.txt file:
Configuration and time taken and Out-of-core:

26. (256, 8) took 3.604ms
27. (384, 5) took 3.602ms
28. (512, 4) took 3.589ms
29. (768, 2) took 3.596ms
30. (1024, 2) took 3.623ms

Since this text file is sorted by destination nodes we use segment scan here. Our segment scan is not properly working (the distance in the output shows up as infinite for all the nodes except for vertex 0). Therefore the output text file and reported times corresponding to it are incorrect.

Nutella04Dest.txt file:
Configuration and time taken and Out-of-core with memory sharing (segscan):

31. (256, 8) took 0.047ms
32. (384, 5) took 0.046ms
33. (512, 4) took 0.05ms
34. (768, 2) took 0.048ms
35. (1024, 2) took 0.05ms

We have used another graph to test and get REPORT RESULTS ON IMPLEMENTATION I:

Bellman-Ford Algorithm is on graph p2p-Gnutella31.txt

from http://snap.stanford.edu/data/p2p-Gnutella31.html .

This file did not start with node 0, therefore we used the sed command provided in the announcements to change the 1s to 0s. On top of that we used sorter.c again to make p2p-Gnutella31 sort by source nodes and by destination nodes and named them Nutella31Source.txt and Nutella31Dest.txt.

p2p-Gnutella31 file:
Configuration and time taken and Incore:

36. (256, 8) took 5.232ms
37. (384, 5) took 5.237ms
38. (512, 4) took 5.252ms
39. (768, 2) took 5.254ms
40. (1024, 2) took 5.238ms

p2p-Gnutella31 file:
Configuration and time taken and Out-of-core:

41. (256, 8) took 13.467ms
42. (384, 5) took 13.568ms
43. (512, 4) took 13.498ms
44. (768, 2) took 13.495ms
45. (1024, 2) took13.354.ms

Default file is not sorted by destination nodes, therefore we do not perform segment scan on it.

Nutella31Source.txt file (Sorted by source nodes):
Configuration and time taken and Incore:

46. (256, 8) took 5.215ms
47. (384, 5) took 5.24ms
48. (512, 4) took 5.227ms
49. (768, 2) took 5.26ms
50. (1024, 2) took 5.236ms

Nutella31Source.txt file:
Configuration and time taken and Out-of-core:

51. (256, 8) took 13.462ms
52. (384, 5) took 13.543ms
53. (512, 4) took 13.528ms
54. (768, 2) took 13.532ms
55. (1024, 2) took 13.357ms

Source sorted file is not sorted by destination nodes, therefore we do not perform segment scan on it..

Nutella31Dest.txt file (Sorted by destination nodes):

Configuration and time taken and Incore:

56. (256, 8) took 5.569ms
57. (384, 5) took 5.608ms
58. (512, 4) took 5.595ms
59. (768, 2) took 5.599ms
60. (1024, 2) took 5.614ms

Nutella31Dest.txt file:

Configuration and time taken and Out-of-core:

61. (256, 8) took 13.16ms
62. (384, 5) took 13.186ms
63. (512, 4) took 13.154ms
64. (768, 2) took 13.164ms
65. (1024, 2) took 13.14ms

Since this text file is sorted by destination nodes we use segment scan here. Our segment scan is not properly working (the distance in the output shows up as infinite for all the nodes except for vertex 0). Therefore the output text file and reported times corresponding to it are incorrect.

Nutella31Dest.txt file:

Configuration and time taken and Out-of-core with memory sharing (segscan):

66. (256, 8) took 0.023ms
67. (384, 5) took 0.023ms
68. (512, 4) took 0.021ms
69. (768, 2) took 0.022ms
70. (1024, 2) took 0.022ms

For the whole of Implementation I, we can see from the above date that Out of core method is significantly slower than incore method (somewhere between 2 ~3 times slower on average). And configurations yield little difference from one another in terms of computation time.

3.2 Pros and Cons of First Implementation

Pros:

- Bellman-Ford algorithm computes the shortest path from a single source vertex to all other vertices in a weighted graph.
- Unlike Djikstra algorithm, Bellman-Ford algorithm is efficient even if there are negative edge weights [4]
- It can be used to quickly detect the presence of negative cycles [4].
- It can also be recast as a dynamic programming algorithm [4].
- Bellman-Ford involves fewer iterations and each iteration is highly parallelizable [3]

Cons:

- The Bellman-Ford algorithm allows to consider all nodes in parallel but for that very reason it is not work efficient [5]. A vertex may be made active multiple times, as a result, an edge may be relaxed several times [3]. Hence, it is not better in terms of work done [3].
- Although Bellman-Ford algorithm can handle negative edge weights, it cannot find the shortest path if there is a negative-weight cycle.It can only detect the negative-weight cycle.
- Even though, the algorithm involves fewer iterations and each iteration is highly parallelizable, the algorithm may process each edge multiple times and is likely to incur high processing time [3]

3.3. REPORT RESULTS ON IMPLEMENTATION II:

WORK EFFICIENT ALGORITHM:

p2p-Gnutella31 file:

Configuration and time taken and Incore:

71. (256, 8) Kernel took 0.787ms, filter took 3.27ms
72. (384, 5) Kernel took 0.795ms, filter took 3.668ms
73. (512, 4) Kernel took 0.783ms, filter took 3.274ms
74. (768, 2) Kernel took 0.791ms, filter took 3.537ms
75. (1024, 2) Kernel took 0.802ms, filter took 3.634ms

p2p-Gnutella31 file:

Configuration and time taken and Out-of-core:

76. (256, 8) Kernel took 0.545ms, filter took 1.177ms
77. (384, 5) Kernel took 0.562ms, filter took 1.204ms
78. (512, 4) Kernel took 0.577ms, filter took 1.537ms
79. (768, 2) Kernel took 0.556ms, filter took 1.295ms
80. (1024, 2) Kernel took 0.566ms, filter took 1.533ms

Nutella31Source.txt file (Sorted by source nodes):

Configuration and time taken and Incore:

81. (256, 8) Kernel took 0.768ms, filter took 3.285ms
82. (384, 5) Kernel took 0.804ms, filter took 3.658ms
83. (512, 4) Kernel took 0.799ms, filter took 3.615ms
84. (768, 2) Kernel took 0.787ms, filter took 3.878ms
85. (1024, 2) Kernel took 0.808ms, filter took 3.637ms

Nutella31Source.txt file:

Configuration and time taken and Out-of-core:

86. (256, 8) Kernel took 0.543ms, filter took 1.183ms
87. (384, 5) Kernel took 0.561ms, filter took 1.238ms
88. (512, 4) Kernel took 0.548ms, filter took 1.184ms
89. (768, 2) Kernel took 0.546ms, filter took 1.286ms
90. (1024, 2) Kernel took 0.543ms, filter took

1.189ms

Nutella31Dest.txt file (Sorted by destination nodes):

Configuration and time taken and Incore:

91. (256, 8) Kernel took 0.753ms, filter took 8.084.ms

92. (384, 5) Kernel took 0.762ms, filter took 8.28ms

93. (512, 4) Kernel took 0.746ms, filter took 8.064ms

94. (768, 2) Kernel took 0.776ms, filter took 8.732ms

95. (1024, 2) Kernel took 0.735ms, filter took 8.014ms

Nutella31Dest.txt file:

Configuration and time taken and Out-of-core:

96. (256, 8) Kernel took 0.535ms, filter took 1.907ms

97. (384, 5) Kernel took 0.526ms, filter took 1.923ms

98. (512, 4) Kernel took 0.524ms, filter took 1.891ms

99. (768, 2) Kernel took 0.531ms, filter took 1.86ms

100. (1024, 2) Kernel took 0.527ms, filter took 1.204ms

Even though there is a filtering stage for Implementation II, since the main focus is on improving performance of the Kernel computation stage and not the performance of the filtering stage, we can observe from date from #36~65 from the implementation I from results report and compare that date to the data from #71~100 from implementation II from results report and see that the average kernel computation time is significantly lower for implementation II. The result is similar when using a different graph text file such as p2pGnutella04.txt. By checking the date from #1~ 30 and comparing that data to the data from #101~130, we can conclude that, indeed Implementation 2 is the more efficient algorithm in terms of Kernel computation time.

We use p2pGnutella04.txt again for Implementation II.

p2p-Gnutella04 file:

Configuration and time taken and Incore:

101. (256, 8) Kernel took 0.36ms, filter took 1.168ms

102.. (384, 5) Kernel took 0.358ms, filter took 1.172ms

103. (512, 4) Kernel took 0.332ms, filter took 1.031ms

104. (768, 2) Kernel took 0.359ms, filter took 1.111ms

105.(1024, 2) Kernel took 0.354ms, filter took 1.166ms

p2p-Gnutella04 file:

Configuration and time taken and Out-of-core:

106. (256, 8) Kernel took 0.235ms, filter took 1.008ms

107. (384, 5) Kernel took 0.28ms, filter took 1.367ms

108. (512, 4) Kernel took 0.234ms, filter took 1.004ms

109. (768, 2) Kernel took 0.225ms, filter took 0.94ms

110. (1024, 2) Kernel took 0.387ms, filter took 2.373ms

Nutella04Source.txt file (Sorted by source nodes):

Configuration and time taken and Incore:

111. (256, 8) Kernel took 0.375ms, filter took

1.181ms

112. (384, 5) Kernel took 0.373ms, filter took 1.175ms

113. (512, 4) Kernel took 0.361ms, filter took 1.055ms

114. (768, 2) Kernel took 0.334ms, filter took 1.345ms

115. (1024, 2) Kernel took 0.347ms, filter took 1.175ms

Nutella04Source.txt file:

Configuration and time taken and Out-of-core:

116. (256, 8) Kernel took 0.201ms, filter took 0.557ms

117. (384, 5) Kernel took 0.213ms, filter took 0.786ms

118. (512, 4) Kernel took 0.225ms, filter took 0.905ms

119. (768, 2) Kernel took 0.226ms, filter took 0.974ms

120. 1024, 2) Kernel took 0.261ms, filter took 1.277ms

Nutella04Dest.txt file (Sorted by destination nodes):

Configuration and time taken and Incore:

121. (256, 8) Kernel took 0.339ms, filter took 2.75ms

122. (384, 5) Kernel took 0.349ms, filter took 2.736ms

123. (512, 4) Kernel took 0.385ms, filter took 3.184ms

124. (768, 2) Kernel took 0.387ms, filter took 3.202ms

125.(1024, 2) Kernel took 0.375ms, filter took 2.954ms

Nutella04Dest.txt file:

Configuration and time taken and Out-of-core:

126. (256, 8) Kernel took 0.215ms, filter took 0.905ms

127. (384, 5) Kernel took 0.176ms, filter took 0.832ms

128. (512, 4) Kernel took 0.185ms, filter took 0.814ms

129. (768, 2) Kernel took 0.175ms, filter took 0.813ms

130. (1024, 2) Kernel took 0.193ms, filter took 0.846ms

Unlike in Implementation I, out of core yields faster computation time than incore does for the whole of Implementation II date.

Though, for configurations, from our data not much variation in results in seen to conclude anything.We were able to conclude from data #91~95 and #121~125, that performing incore on destination sorted files yields high kernel and very high filtration times. From data #86~90 and #116~120 we are able to conclude that performing out-of-core on source sorted files yields relatively low kernel and low filtration times.

3.5. Pros and Cons of Work Efficient Algorithm:

Pro:

Kernel computation was significantly lower than in Implementation I. We achieved high speedups and improved performance of kernel computation.

Con:

Filtration time can sometimes be unexpectedly high

4. CONCLUSION:

In this project, we have demonstrated the effectiveness of parallelizing single source shortest path (SSSP) algorithm on GPUs for different graphs having large datasets. We

presented two parallel implementations based on Bellman Ford algorithm on GPU using CUDA parallel programming interface. We ran the two algorithms on various large graphs and checked for improving performance of the computation stage.The fine-grained, parallel primitives are used to reorganize data in the second version (Work Efficient algorithm). As expected, we achieved high speedups in the kernel computation and improved the performance of the computation stage, as was the focus of this project, in the second version of parallel Bellman Ford algorithm under Work Efficient algorithm. Furthermore, the Work Efficient method showed better parallelism as well as work-efficiency than the first implementation of Bellman-Ford algorithm.

REFERENCES:

[1] 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. Web. 10 Apr. 2017. <http://escholarship.org/uc/item/8qr166v2#page-1>

[2] Agarwal, Pankhari, and Maitreyee Dutta. "New Approach of Bellman Ford Algorithm on GPU using Compute Unified Design Architecture (CUDA)." Research.ijcaonline.org. International Journal of Computer Applications Volume 110, Jan. 2015. Web. 10Apr.2017. <http://research.ijcaonline.org/volume110/number13/pxc3901027.pdf>.

[3] Chakaravarthy, Venkatesan T., Fabio Checconi, Fabrizio Petrini, and Yogish Sabharwal. "Scalable Single Source Shortest Path Algorithms for Massively Parallel Systems." Pdfs.semanticscholar.org. N.p., n.d. Web.10Apr.2017. <https://pdfs.semanticscholar.org/dad4/0a530c4a495fdca711d43f20b90ed1824550.pdf>.

[4] Erickson, Jeff. "Shortest Paths." Courses.engr.illinois.edu. Creative Commons License, n.d.Web.10Apr.2017. <https://courses.engr.illinois.edu/cs473/sp2017/notes/08-sssp.pdf>.

[5] Meyer, U., and P. Sanders. "-Stepping : A Parallel Single Source Shortest Path Algorithm." Cs.utexas.edu. Max-Planck-Institut fur Informatik, Germany, n.d. Web. 10Apr.2017. <https://www.cs.utexas.edu/~pingali/CS395T/2013fa/papers/delta-stepping.pdf>.

[6] Papaefthymiou, Marios, and Joe Rodrigue. "Implementing Parallel Shortest Paths Algorithms." Citeseerx.ist.psu.edu. Department of Computer Science, Yale University, n.d. Web. 10 Apr. 2017. <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.49.6379&rep=rep1&type=pdf>.

[7] Singh, Dhirendra Pratap , and Nilay Khare. "Parallel Implementation of the Single Source Shortest Path Algorithm on CPU-GPU Based Hybrid System." Academia.edu. International Journal of Computer Science and Information Security, Sept. 2013. Web. 10 Apr.2017. <http://www.academia.edu/11928177/Parallel_Implementation_of_the_Single_Source_Shortest_Path_Algorithm_on_CPU_GPU_Based_Hybrid_System>.