Parallelizing Single Source Shortest Path (SSSP) on GPU

Olaolu Emmanuel, *Rutgers University*

Abstract

In this project, we are exploring different methods of calculating the single source shortest path by exploiting the large amount of threads available to us by the GPU. To fully take advantage of these threads, we will design our kernels to maximize the amount of active threads at any given point in time. We start by choosing a well known algorithm for calculating the single source shortest path, which lends itself to parallelism, and adding optimizations to further reduce the overall running time.

1. Bellman-ford Algorithm

1.1. Introduction

While this algorithm does not have an amazing runtime, it does have the advantage of being fairly simple to parallelize. This is because each edge is processed independently of the others which makes it easy to distribute the work amongst threads without having to worry about any of the common parallel programming issues such as race conditions.

1.2. Algorithm

The Bellman-ford Algorithm works by keeping two vectors of the distance from the source node to the node at index i. One vector represents the previous distances and the other represents the current. It then iterates through all the edges and updates any distances if it finds a lower one. After one iteration, a check is made to see if any distances have changed and the loop breaks if there have been none.

1.3. Program Design

This program has two main components, a kernel which processes the edges and a host method which continuously calls the kernel until it reaches an iteration where no changes have been made to the distance array. The host method is also responsible for swapping the pointers to the previous and current distance array vectors in between calls to the kernel. The GPU used for these computations is the GeForce GT 630.

1.4. Execution Configuration

amazon0312.txt(http://snap.stanford.edu/data/amazon0312.html):

(256, 8): 488.862000 milli-seconds
(384, 5): 487.697000 milli-seconds
(512, 4): 490.263000 milli-seconds
(768, 2): 488.876000 milli-seconds
(1024, 2): 480.428000 milli-seconds

msdoor.txt(Florida sparse matrix collection):

(256, 8): 34940.103000 milli-seconds
(384, 5): 34930.628000 milli-seconds
(512, 4): 34950.181000 milli-seconds
(768, 2): 35592.102000 milli-seconds
(1024, 2): 34909.235000 milli-seconds

road-cal.txt(http://www.dis.uniroma1.it/challenge9/download.shtml):

(256, 8): 46394.053000 milli-seconds
(384, 5): 46233.056000 milli-seconds
(512, 4): 46475.115000 milli-seconds
(768, 2): 46675.861000 milli-seconds
(1024, 2): 45975.054000 milli-seconds

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

(256, 8): 6268.452000 milli-seconds
(384, 5): 6247.264000 milli-seconds
(512, 4): 6246.831000 milli-seconds
(768, 2): 6585.056000 milli-seconds
(1024, 2): 6213.650000 milli-seconds

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

(256, 8): 1264.867000 milli-seconds
(384, 5): 1027.450000 milli-seconds
(512, 4): 1018.434000 milli-seconds
(768, 2): 1027.676000 milli-seconds
(1024, 2): 1035.385000 milli-seconds

1.6. Edge List Organization

Sorted by source:

amazon0312.txt(http://snap.stanford.edu/data/amazon0312.html):

(256, 8): 488.862000 milli-seconds
(384, 5): 487.697000 milli-seconds
(512, 4): 490.263000 milli-seconds
(768, 2): 488.876000 milli-seconds
(1024, 2): 480.428000 milli-seconds

msdoor.txt(Florida sparse matrix collection):

(256, 8): 34940.103000 milli-seconds
(384, 5): 34930.628000 milli-seconds
(512, 4): 34950.181000 milli-seconds
(768, 2): 35592.102000 milli-seconds
(1024, 2): 34909.235000 milli-seconds

road-cal.txt(http://www.dis.uniroma1.it/challenge9/download.shtml):

(256, 8): 46394.053000 milli-seconds
(384, 5): 46233.056000 milli-seconds
(512, 4): 46475.115000 milli-seconds
(768, 2): 46675.861000 milli-seconds
(1024, 2): 45975.054000 milli-seconds

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

(256, 8): 6268.452000 milli-seconds
(384, 5): 6247.264000 milli-seconds
(512, 4): 6246.831000 milli-seconds
(768, 2): 6585.056000 milli-seconds
(1024, 2): 6213.650000 milli-seconds

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

(256, 8): 1264.867000 milli-seconds
(384, 5): 1027.450000 milli-seconds
(512, 4): 1018.434000 milli-seconds
(768, 2): 1027.676000 milli-seconds
(1024, 2): 1035.385000 milli-seconds

Sorted by destination:

amazon0312.txt(http://snap.stanford.edu/data/amazon0312.html):

(256, 8): 773.135000 milli-seconds
(384, 5): 778.125000 milli-seconds
(512, 4): 769.078000 milli-seconds
(768, 2): 960.814000 milli-seconds
(1024, 2): 766.689000 milli-seconds

msdoor.txt(Florida sparse matrix collection):

(256, 8): 35530.935000 milli-seconds
(384, 5): 35499.054000 milli-seconds
(512, 4): 35950.082000 milli-seconds
(768, 2): 35412.10000 milli-seconds
(1024, 2): 35080.110000 milli-seconds

road-cal.txt(http://www.dis.uniroma1.it/challenge9/download.shtml):

(256, 8): 48551.695000 milli-seconds
(384, 5): 48506.495000 milli-seconds
(512, 4): 48710.230000 milli-seconds
(768, 2): 48655.23300 milli-seconds
(1024, 2): 48331.113000 milli-seconds

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

(256, 8): 7143.562000 milli-seconds
(384, 5): 7215.393000 milli-seconds
(512, 4): 7158.967000 milli-seconds
(768, 2): 7209.273000 milli-seconds
(1024, 2): 7303.883000 milli-seconds

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

(256, 8): 1264.867000 milli-seconds
(384, 5): 1027.450000 milli-seconds
(512, 4): 1552.032000 milli-seconds
(768, 2): 1475.940000 milli-seconds
(1024, 2): 1498.449000 milli-seconds

1.8. In-core Configuration

Sorted by source:

amazon0312.txt(http://snap.stanford.edu/data/amazon0312.html):

(256, 8): 309.473000 milli-seconds
(384, 5): 312.448000 milli-seconds
(512, 4): 306.437000 milli-seconds
(768, 2): 298.050000 milli-seconds
(1024, 2): 331.508000 milli-seconds

msdoor.txt(Florida sparse matrix collection):

(256, 8): 14989.633000 milli-seconds
(384, 5): 14861.328000 milli-seconds
(512, 4): 14928.618000 milli-seconds

(768, 2): 14787.045000 milli-seconds
(1024, 2): 14833.278000 milli-seconds
road-cal.txt(http://www.dis.uniroma1.it/challenge9/download.shtml):
(256, 8): 29132.103000 milli-seconds
(384, 5): 29102.456000 milli-seconds
(512, 4): 29455.013000 milli-seconds
(768, 2): 29511.201000 milli-seconds
(1024, 2): 29707.634000 milli-seconds
roadNet-CA.txt(http://snap.stanford.edu/data/roadNet-CA.html):
(256, 8): 6268.452000 milli-seconds
(384, 5): 6247.264000 milli-seconds
(512, 4): 6246.831000 milli-seconds
(768, 2): 6585.056000 milli-seconds
(1024, 2): 6213.650000 milli-seconds
web-Google.txt(http://snap.stanford.edu/data/web-Google.html):
(256, 8): 4037.613000 milli-seconds
(384, 5): 4029.755000 milli-seconds
(512, 4): 4051.060000 milli-seconds
(768, 2): 4099.970000 milli-seconds
(1024, 2): 3970.179000 milli-seconds

Sorted by destination:

amazon0312.txt(http://snap.stanford.edu/data/amazon0312.html):
(256, 8): 793.600000 milli-seconds
(384, 5): 493.097000 milli-seconds
(512, 4): 695.782000 milli-seconds
(768, 2): 680.967000 milli-seconds
(1024, 2): 479.883000 milli-seconds
msdoor.txt(Florida sparse matrix collection):
(256, 8): 15100.494000 milli-seconds
(384, 5): 14261.113000 milli-seconds
(512, 4): 14903.422000 milli-seconds
(768, 2): 14905.555000 milli-seconds
(1024, 2): 15043.938000 milli-seconds
road-cal.txt(http://www.dis.uniroma1.it/challenge9/download.shtml):
(256, 8): 27406.451000 milli-seconds
(384, 5): 27672.237000 milli-seconds
(512, 4): 27136.223000 milli-seconds
(768, 2): 27147.352000 milli-seconds
(1024, 2): 27133.451000 milli-seconds
roadNet-CA.txt(http://snap.stanford.edu/data/roadNet-CA.html):
(256, 8): 4739.248000 milli-seconds
(384, 5): 4521.447000 milli-seconds
(512, 4): 4555.934000 milli-seconds
(768, 2): 4502.490000 milli-seconds
(1024, 2): 4511.904000 milli-seconds
web-Google.txt(http://snap.stanford.edu/data/web-Google.html):
(256, 8): 1057.281000 milli-seconds
(384, 5): 1133.609000 milli-seconds
(512, 4): 1055.503000 milli-seconds
(768, 2): 1052.742000 milli-seconds
(1024, 2): 1032.815000 milli-seconds

1.10. Pros

- This algorithm lends itself to parallelism which makes it easier to translate to the GPU. This is because edges are processed independently of one another which makes distributing work amongst threads much simpler.

1.11. Cons

- There could be some thread divergence when checking to see if an update needs to be made to the distance vector.

2. Work Efficient Algorithm

2.1. Pros

- This algorithm can increase the speed of the edge processing considerably over time since the input amount of edges is reduced every iteration.

2.2. Cons

- There is the added amount of work of having to iterate though the to-process edges and copy them over to the new to-process vector. This can be expensive, especially towards the beginning of the run where almost all edges are in the to-process vector.

2.3. Execution Configuration

amazon0312.txt(http://snap.stanford.edu/data/amazon0312.html):
(256, 8): 460.722000 milli-seconds

(384, 5): 458.364000 milli-seconds
(512, 4): 456.925000 milli-seconds
(768, 2): 471.213000 milli-seconds
(1024, 2): 461.134000 milli-seconds

msdoor.txt(Florida sparse matrix collection):

(256, 8): 47212.433000 milli-seconds
(384, 5): 47921.213000 milli-seconds
(512, 4): 48691.578000 milli-seconds
(768, 2): 48112.400000 milli-seconds
(1024, 2): 48109.350000 milli-seconds

road-cal.txt(http://www.dis.uniroma1.it/challenge9/download.shtml):

(256, 8): 56241.507000 milli-seconds
(384, 5): 56123.446000 milli-seconds
(512, 4): 56020.485000 milli-seconds
(768, 2): 56294.304000 milli-seconds
(1024, 2): 56157.014000 milli-seconds

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

(256, 8): 12047.604000 milli-seconds
(384, 5): 11942.144000 milli-seconds
(512, 4): 12342.845000 milli-seconds
(768, 2): 12525.156000 milli-seconds
(1024, 2): 11809.045000 milli-seconds

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

(256, 8): 857.911000 milli-seconds
(384, 5): 811.650000 milli-seconds
(512, 4): 825.294000 milli-seconds
(768, 2): 894.147000 milli-seconds
(1024, 2): 832.955000 milli-seconds

2.4. Computation Filtering

***All tests run using 8 blocks of size 256 (incore)**

amazon0312.txt(http://snap.stanford.edu/data/amazon0312.html):

computation: 0.090000 seconds
filtering: 0.246000 seconds

msdoor.txt(Florida sparse matrix collection):

computation: 2.344001 seconds
filtering: 6.01803 seconds

road-cal.txt(http://www.dis.uniroma1.it/challenge9/download.shtml):

computation: 9.477795 seconds
filtering: 24.297434 seconds

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

computation: 31.514001 seconds
filtering: 90.132803 seconds

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

computation: 0.098001 seconds
filtering: 0.35167 seconds

2.5. Data Analysis

Dest vs. Src Sorting

It seems that the quickest run is still the one which uses the most threads per block. However, something interesting to note is that almost every run of the algorithm sorted by destination is slower than the one sorted by source. This may have to do with the fact that threads in the same block or more likely to be updating the same index since it is sorted by destination. However, this is only an assumption and unfortunately I do not have any statistics to support this claim.

Incore vs. Outcore

It seems that sorting by source yields better runtimes yet again, but I found something else to be interesting in this data. The performance of the in-core version seems to be ~33% better than that of the out-core version. I believe that this is due to memory coalescing since the memory addresses being accessed by the different threads all belong to the same vector which should increase the chance of them being coalesced[1].

Block Size

Judging by these tests, it seems that this algorithm performs best with larger block sizes as opposed to a larger number of blocks. This could be due to a block number limit on the GPU that would result in excess blocks having to wait in a queue before they are executed.

3. References

1.https://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/