Project2 : SSSP on GPU

Sairama Rachakonda
Rutgers University

Abstract

This report presents analysis of some implementations for single source shortest path algorithm on GPU. First implementation tries to make bellman fords sequential algorithm, parallel and analyze 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 Data Set Used

The following list of graphs have been used for testing our implementations.

- Road Net CA
- Web- Google
- Amazon0312
- msdoor
- road-cal
- Live Journal

The Live Journal graph hasn't been used in the implementation as,it was very huge, and hence throwing segmentation fault on the host itself. The below Table-1 gives the property of the graphs:

	No. of Edges	Diameter
roadNet-CA	2M	187
ms-door	20.65M	167
road-Cal	4.6M	2575
web-Google	5M	21
amazon0312	3M	18

2 Implementation-1 using Bellman Ford algorithm

2.1 Algorithm

Bellman ford algorithm is suitable for parallel implementation. The sequential bellman-ford algorithm verifies 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) where V and E are the number of vertices and edges of the graph respectively. In the parallel bellman-ford algorithm, we try to utilize the parallelism of edge processing in each iteration. Every input edge is verified at every iteration. Hence the edges are distributed to different threads. This distribution is done with a good engineering such that, the load is balanced evenly for better parallelism.

2.2 Outcore implementation

In the Outcore implementation, the result of one iteration is used only in the next iteration. The below are the results of the Outcore implementation for various block configurations.

	Outcore 1024*8		
	Input	Source	Destination
roadNet-CA	7s 75ms	6s 767ms	6s 758ms
ms-door	24s 134ms	24s 858ms	24s 819ms
road-Cal	50s 852ms	46s 880ms	46s 233ms
web-Google	770ms	914ms	946ms
amazon0312	451ms	452ms	483ms

	Out-core 768*2		
	Input	Source	Destination
roadNet-CA	6s 984ms	6s 702ms	6s 699ms
ms-door	24s 318ms	25s 105ms	25s 58ms
road-Cal	50s 254ms	46s 455ms	45s 746ms
web-Google	750ms	903ms	940ms
amazon0312	447ms	448ms	477ms

	Out-core 512*4		
	Input	Source	Destination
roadNet-CA	7s 732ms	7s 463ms	7s 51ms
ms-door	25s 83ms	25s 52ms	25s 210ms
road-Cal	52s 224ms	47s 964ms	47s 59ms
web-Google	773ms	921ms	951ms
amazon0312	550ms	501ms	533ms

	Out-core 384*5		
	Input	Source	Destination
roadNet-CA	7s 246ms	6s 989ms	6s 987ms
ms-door	24s 983ms	25s 711ms	25s 750ms
road-Cal	52s 373ms	48s 394ms	47s 635ms
web-Google	772ms	917ms	948ms
amazon0312	461ms	462ms	489ms

	Outcore 256*8		
	Input	Source	Destination
roadNet-CA	7s 183ms	6s 933ms	6s 947ms
ms-door	24s 485ms	25s 249ms	25s 322ms
road-Cal	51s 182ms	47s 470ms	47s 60ms
web-Google	762ms	922ms	951ms
amazon0312	457ms	458ms	483ms

2.3 In-core implementation

In the in-core implementation,unlike the out-core, the change made to the distance of a vertex, is available in the same iteration and not in the next iteration. The following are the in-core implementation results.

	Incore 1024*8		
	Input	Source	Destination
roadNet-CA	4s 132ms	4s 394ms	4s 358ms
ms-door	7s 912ms	8s 662ms	8s 662ms
road-Cal	34s 187ms	28s 719ms	28s 236ms
web-Google	407ms	510ms	472ms
amazon0312	156ms	167ms	166ms

	Incore 768*2		
	Input	Source	Destination
roadNet-CA	4s 162ms	4s 277ms	4s 301ms
ms-door	7s 706ms	8s 473ms	8s 529ms
road-Cal	34s 171ms	28s 555ms	28s 165ms
web-Google	444ms	506ms	498ms
amazon0312	145ms	135ms	165ms

	Incore 512*4		
	Input	Source	Destination
roadNet-CA	4s 509ms	4s 545ms	4s 519ms
ms-door	8s 373ms	8s 776ms	8s 968ms
road-Cal	35s 158ms	29s 205ms	28s 436ms
web-Google	408ms	539ms	445ms
amazon0312	157ms	157ms	167ms

	Incore 384*5		
	Input	Source	Destination
roadNet-CA	4s 300ms	4s 502ms	4s 434ms
ms-door	7s 780ms	8s 999ms	8s 951ms
road-Cal	35s 872ms	30s 79ms	29s 910ms
web-Google	435ms	486ms	529ms
amazon0312	160ms	160ms	170ms

	Incore 256*8		
	Input	Source	Destination
roadNet-CA	4s 180ms	4s 463ms	4s 424ms
ms-door	7s 990ms	8s 798ms	8s 843ms
road-Cal	34s 880ms	29s 243ms	28s 883ms
web-Google	428ms	489ms	448ms
amazon0312	159ms	159ms	168ms

2.4 Shared memory with Outcore implementation

For this implementation we would like to use the shared memory to perform the minimum of calculated distance of a destination vertex and the its current distance. We create a shared memory equal to the number of threads in a block. The shared memory stores the destination vertex of the edge it handles. Also the shared memory stores the calculated distance of the destination vertex. Now we use a segmented scan like approach from our Project-1 to find the minimum for each destination vertex. The below are the results for shared memory. To perform a segmented scan over the edges, we sort the edges in the destination form.

			Shared Memory		
	1024*2	768*2	512*4	384*5	256*8
roadNet-CA	11s 419ms	12s 313ms	10s 400ms	10s 268ms	13s 791ms
ms-door	47s 391ms	49s 576ms	43s 633ms	43s 346ms	55s 905ms
road-Cal	1m 14s 456ms	1m 20s 764ms	1m 7s 970ms	1m 7s 242ms	1m 28s 944ms
web-Google	1s 168ms	1s 196ms	1s 87ms	1s 72ms	1s 251ms
amazon0312	728ms	715ms	640ms	633ms	780ms

3 Analysis of Implementation 1:

- Table 1 shows the diameter of each of the graphs. If the diameter of a graph A is large compared to another graph B, then processing time of graph A should be greater than processing time of graph B. Our results follow this. ROAD-CAL whose diameter is the maximum has maximum processing time across all the configurations. Amazon0312 has the minimum diameter and least running time.

- In-core takes less time than Out-core. Hence, the number of iterations are less for in-core than out-core. The reason which I am predicting is that, in in-core the threads get the updated vertex distance as soon as they get updated in the same iteration, rather than waiting for the next iteration.

- Shared memory implementation takes more time compared to both the in-core and out-core implementations. This happened in the earlier assignment also. We want to save on atomic operation overhead and hence we use segmented scan approach. But due the thread divergence, where some threads can execute certain depth of the segmented scan and others may not be executing. The cost of the thread divergence is more than the overhead cost of atomic operations.

- For ROAD-CAL and MS-Door, the destination sorted approach takes less time than the source sorted approach in the Out-core implementation. The reason can be that, when the edges are sorted by destination, then the neighboring threads of warp try to update the same destination vertex on which they are sorted and hence takes more time.

- The block configuration of 768*2 = 1536, takes less time than the block configurations of 1024*2 =2048 and 512*4=2048. Even though the second block configuration has more threads, it takes less time than the 1536 threads.

- On an average the 768*2 = 1536 configuration executes faster than all the configurations. This may be attributed to the registers and cache which the threads share.

4 Implementation2-Work Efficient

In this approach we try to organize the edges to increase the performance. We try to get inspiration from the Work Front Sweep method discussed in paper:- Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths by Baxter Sean et.al

4.1 Algorithm

In implementation 1 every edge in each iteration irrespective of whether the source of the edge has been changed in the earlier iteration or not. In this implementation, we take this clue and only work on those edges whose source has been changed in the earlier iteration. For an edge with source as u and destination as v, if the distance[u] didn't change, then it is redundant to perform distance[u] + weight ¡ distance[v]. We have three steps

- Warp Count Kernel:- For each warp, counts how many of the edges in the warp had their source vertex distance from the source changed.

- Performs the exclusive prefix sum on the warp count.

- Reorganizes the edges of each warp for the next iteration.

The below are the results of this implementation. Here I specify two timings- Filtering time and processing time. Filtering time is the time taken for the three stages of reorganization. Processing time is the time taken by the kernel which updates the distance from the source (bellman ford).

First I report all the Outcore filtering and processing times.

4.2 Outcore Filtering and Processing Results

		Outcome-Filtering(1024*2)	
	Input	Source	Destination
roadNet-CA	8s 93ms	6s 472ms	10s 538ms
ms-door	29s 687ms	25s 457ms	37s 97ms
road-Cal	1m 0s 181ms	41s 678ms	1m 13s 689ms
web-Google	423ms	329ms	1s 722ms
amazon0312	268ms	268ms	835ms

		Outcore—Filtering(768*2)	
	Input	Source	Destination
roadNet-CA	8s 283ms	7s 426ms	10s 316ms
ms-door	31s 837ms	28s 50ms	37s 367ms
road-Cal	1m 0s 151ms	47s 574ms	1m 12s 291ms
web-Google	439ms	372ms	1s 627ms
amazon0312	301ms	302ms	817ms

		Outcore—Filtering(512*4)	
	Input	Source	Destination
roadNet-CA	8s 86ms	6s 458ms	10s 691ms
ms-door	29s 703ms	25s 577ms	25s 577ms
road-Cal	1m 0s 669ms	41s 467ms	1m 14s 427ms
web-Google	426ms	328ms	1s 722ms
amazon0312	266ms	267ms	838ms

		Outcore—Filtering(384*5)	
	Input	Source	Destination
roadNet-CA	8s 10ms	6s 645ms	10s 549ms
ms-door	29s 908ms	25s 796ms	36s 638ms
road-Cal	1m 0s 808ms	42s 553ms	1m 14s 178ms
web-Google	423ms	334ms	1s 702ms
amazon0312	273ms	273ms	840ms

		Outcore—Filtering(256*8)	
	Input	Source	Destination
roadNet-CA	8s 44ms	6s 461ms	10s 554ms
ms-door	29s 721ms	25s 393ms	36s 523ms
road-Cal	1m 0s 367ms	41s 535ms	1m 13s 959ms
web-Google	423ms	329ms	1s 720ms
amazon0312	267ms	266ms	839ms

4.3 Out-core Processing Results

		Outcore-Processing(1024*2)	
	Input	Source	Destination
roadNet-CA	762ms	759ms	770ms
ms-door	3s 289ms	2s 894ms	3s 472ms
road-Cal	13s 332ms	12s 928ms	12s 57ms
web-Google	66ms	81ms	84ms
amazon0312	37ms	37ms	45ms

		Outcore-Processing(768*2)	
	Input	Source	Destination
roadNet-CA	764ms	764ms	773ms
ms-door	3s 341ms	2s 972ms	3s 555ms
road-Cal	13s 257ms	12s 885ms	12s 74ms
web-Google	63ms	78ms	81ms
amazon0312	36ms	37ms	45ms

		Outcore-Processing(512*4)	
	Input	Source	Destination
roadNet-CA	762ms	761ms	772ms
ms-door	3s 237ms	2s 926ms	2s 926ms
road-Cal	13s 489ms	13s 136ms	12s 209ms
web-Google	70ms	80ms	86ms
amazon0312	37ms	38ms	52ms

		Outcore—Filtering(384*5)	
	Input	Source	Destination
roadNet-CA	8s 10ms	6s 645ms	10s 549ms
ms-door	29s 908ms	25s 796ms	36s 638ms
road-Cal	1m 0s 808ms	42s 553ms	1m 14s 178ms
web-Google	423ms	334ms	1s 702ms
amazon0312	273ms	273ms	840ms

		Outcore-Processing(256*8)	
	Input	Source	Destination
roadNet-CA	762ms	760ms	770ms
ms-door	3s 318ms	2s 910ms	3s 546ms
road-Cal	13s 412ms	13s 107ms	12s 225ms
web-Google	71ms	80ms	86ms
amazon0312	37ms	37ms	54ms

4.4 In-core Filtering Results

		Incore-Filtering(1024*2)	
	Input	Source	Destination
roadNet-CA	8s 78ms	6s 460ms	10s 525ms
ms-door	28s 855ms	24s 777ms	35s 980ms
road-Cal	1m 0s 41ms	41s 307ms	401ms
web-Google	401ms	322ms	1s 566ms
amazon0312	269ms	267ms	733ms

		Incore-Filtering(768*2)	
	Input	Source	Destination
roadNet-CA	8s 258ms	7s 439ms	10s 339ms
ms-door	31s 79ms	27s 501ms	36s 237ms
road-Cal	1m 0s 216ms	47s 595ms	1m 12s 469ms
web-Google	399ms	352ms	1s 632ms
amazon0312	302ms	304ms	698ms

		Incore-Filtering(512*4)	
	Input	Source	Destination
roadNet-CA	8s 144ms	6s 491ms	10s 740ms
ms-door	28s 631ms	24s 400ms	35s 290ms
road-Cal	1m 0s 240ms	41s 408ms	1m 14s 84ms
web-Google	399ms	318ms	1s 562ms
amazon0312	267ms	267ms	734ms

		Incore-Filtering(384*5)	
	Input	Source	Destination
roadNet-CA	8s 32ms	6s 647ms	10s 559ms
ms-door	29s 41ms	25s 370ms	35s 590ms
road-Cal	1m 0s 479ms	42s 219ms	1m 13s 974ms
web-Google	398ms	324ms	1s 648ms
amazon0312	273ms	272ms	740ms

		Incore-Filtering(256*8)	
	Input	Source	Destination
roadNet-CA	8s 49ms	6s 464ms	10s 564ms
ms-door	28s 414ms	24s 215ms	34s 779ms
road-Cal	1m 0s 181ms	41s 240ms	1m 13s 424ms
web-Google	385ms	319ms	1s 558ms
amazon0312	267ms	267ms	736ms

4.5 In-core Processing Results

		Incore-Processing(1024*2)	
	Input	Source	Destination
roadNet-CA	332ms	344ms	353ms
ms-door	1s 691ms	1s 332ms	2s 184ms
road-Cal	6s 242ms	1m 13s 478ms	55ms
web-Google	55ms	75ms	75ms
amazon0312	38ms	30ms	49ms

		Incore-Processing(768*2)	
	Input	Source	Destination
roadNet-CA	348ms	348ms	357ms
ms-door	1s 731ms	1s 419ms	2s 179ms
road-Cal	6s 314ms	5s 703ms	5s 733ms
web-Google	58ms	66ms	75ms
amazon0312	30ms	29ms	42ms

		Incore-Processing(512*4)	
	Input	Source	Destination
roadNet-CA	345ms	346ms	360ms
ms-door	1s 607ms	1s 345ms	2s 39ms
road-Cal	6s 344ms	5s 785ms	5s 776ms
web-Google	58ms	67ms	78ms
amazon0312	30ms	30ms	41ms

		Incore-Processing(384*5)	
	Input	Source	Destination
roadNet-CA	342ms	343ms	353ms
ms-door	1s 495ms	1s 354ms	2s 82ms
road-Cal	6s 274ms	5s 857ms	5s 786ms
web-Google	63ms	62ms	80ms
amazon0312	30ms	31ms	42ms

		Incore-Processing(256*8)	
	Input	Source	Destination
roadNet-CA	347ms	347ms	355ms
ms-door	1s 692ms	1s 493ms	2s 58ms
road-Cal	6s 357ms	5s 829ms	5s 782ms
web-Google	52ms	72ms	72ms
amazon0312	30ms	30ms	41ms

5 Analysis of Implementation 2:

- The processing time is less than the filtering time for each execution. This is obvious because in filtering stage we perform the reorganization to make the processing time lesser. We take the overhead of reorganization.

- Like in implementation 1, the out-core processing time is more than the in-core processing time, for the same reason mentioned in the analysis of implementation 1.

- The filtering time is almost the same for both out-core and incore implementations. This is obvious because,the filtering time depends on the number of edges for which the source distance changes. For in-core and out-core at the end of their distance verification steps, number of edges changed will be the same.

- One major observation is that, if the edge list is sorted by the source vertex, the filtering time remarkably less when sorted with destination. Multiple thoughts came for me to reason this.

  - First thing is because of more contention for atomic instruction, when it is sorted by destination. But this doesn't seem valid to me because, in the implementation we use atomic min while getting the warp count. This array is of size 64. So its not a big deal for 64 sized array. Also, we use atomic min for the exclusive prefix sum. But this is also a 64 size array. This is independent of whether the edge list is sorted by source or destination.

  - So, what I felt is that some difference is happening in the stage, where we are reordering the vertices (Third kernel).

- The filtering time is very high for the large graphs. Its almost as much as the running time in Implementation 1.

6 Implementation -3

Here,I tried to implement the Near-far pile method discussed in the paper Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths. In the earlier implementation, we didn't prioritize any vertices which need to be processed next. In this implementation, I tried to create a Near and Far pile, with delta*i as the splitting parameter the pile. All the vertices which are less than delta*i will go to the Near pile. All the vertices with greater value go to the far pile. On the near pile I execute the in-core implementation. For those vertices which have been updated, if their distance from the source is less than splitting distance, they fall in Near pile, else in far pile. I once again apply incore on the Near pile until, it gets empty. Then I take the far pile, and increase the splitting distance to delta*(i+1). Once again I repeat the procedure.

But, I couldn't fetch the results, as the implementation gave few errors.

7 How to Run

To run the code use the scripts provided in the folder.