

Summary of Message Spreading Simulation

CUDA stage 4.

1. Problem analysis

Simulation of message spreading in social networks.



source: https://en.wikipedia.org/wiki/Network_theory#/media/File:Social_Network_Analysis_Visualization.png

The problem might be important from the point of view of sociology and viral marketing¹.

A result of the target program should answer following questions:

- How many people receive the message?
- How looks the structure of message spreading in the network?

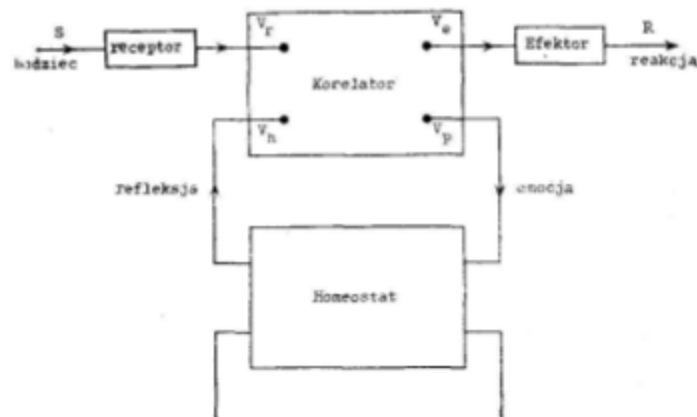
2. Solution description

Following theories have been used to model the problem:

- graph theory to describe a system, where nodes mean elementary objects (people) and edges mean relationships between them (get to know someone relationship, X get to know Y, Y get to know X),

¹ https://en.wikipedia.org/wiki/Viral_marketing https://en.wikipedia.org/wiki/Viral_video

- autonomic systems theory² to algorithmize a human psyche.



Rys.10.1 Obszar informacyjny systemu autonomicznego

source: Mazur M., Cybernetyka i charakter. Wyd. 1, PIW, Warsaw 1976, page 156.

Assumptions:

1. Source is spreading one type of message at the beginning of simulation.
2. Node passes a received message only once to all its neighbour, i.e. if a node already passed a message then it does not pass messages later.
3. Due to a short time of the simulation for typical cases (max. a few days) one can assume that there is no unregistration of conductivity.

Input parameters:

1. Initial reflection potential $v_h^{(0)}$,
2. Conductivity G_0 and G_{max} ,
3. Message passing time t_c and message processing time t_p ,
4. Decision-making potential v_d ,
5. Total simulation time t_s ,
6. Graph $G(V, E)$,
7. Source vertex, which broadcasts first messages (e.g. facebook profile).

Schema of algorithm:

initialization:

1. Read input data.
2. Create initial messages vector $M^{(0)}$ (of size $|E|$) based on source vertex and graph G .

for each node:

1. Get messages to process from M vector and sort them by arrival time.
2. Process messages.
3. If computation indicates that message should be pass then send a message to all neighbours and finish.

² Mazur M., Cybernetyka i charakter. Wyd. 1, PIW, Warszawa 1976.

4. Finish if the simulation time is over.
5. Go back to step 1.

Output:

1. Graph $G(V, E, M')$ in dot format³ which contains information how the messages was passing in a network during simulation.

According to many papers^{[4][5]} a forward star is a recommended graph representation in CUDA.

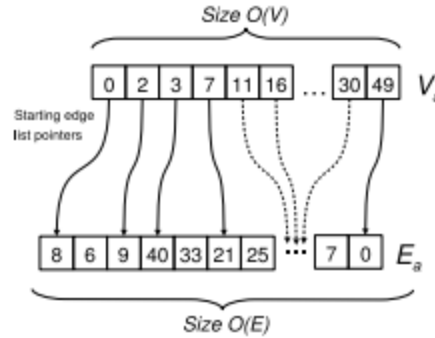


Fig. 3. Graph representation with vertex list pointing to a packed edge list.

source: P. Harish, P. J. Narayanan, Accelerating large graph algorithms on the GPU using CUDA, page 5.

3. Input parameters and file format

```
./msg-spr-sim [<no_of_threads_per_block>] [<device_id>] <"<input_file>">
<"<output_file>" 2>"<file_with_exec_time>"
```

Default value of:

- *no_of_threads_per_block* is the maximum number of threads per block supported by the device,
- *device_id* is 0, i.e. first available device.

Input file format:

```

| %d                // number of vertices
x #V | %f %f %f %f  // v_h_i, G_0_i, G_max_i, v_d_i
| %d                // number of edges
x #V | %d [%d, ...]  // number of edges of vertex v_i [v_j, ...]
| %d                // source vertex v_i
| %d %d %d          // t_c, t_p, t_s
```

³ [https://en.wikipedia.org/wiki/DOT_\(graph_description_language\)](https://en.wikipedia.org/wiki/DOT_(graph_description_language))

⁴ D. Merrill, M. Garland, A. Grimshaw, Scalable GPU Graph Traversal, [access 05/01/2015], <https://research.nvidia.com/sites/default/files/publications/ppo213s-merrill.pdf>

⁵ P. Harish, P. J. Narayanan, Accelerating large graph algorithms on the GPU using CUDA, [access 05/01/2015], <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.102.4206&rep=rep1&type=pdf>

Example:

```
5
0 0 0 0
1 1 10 5
1 1 10 5
0 1 10 5
-1 1 10 5
10
2 1 2
2 0 3
2 0 3
3 1 2 4
1 3
0
3 30 330
```

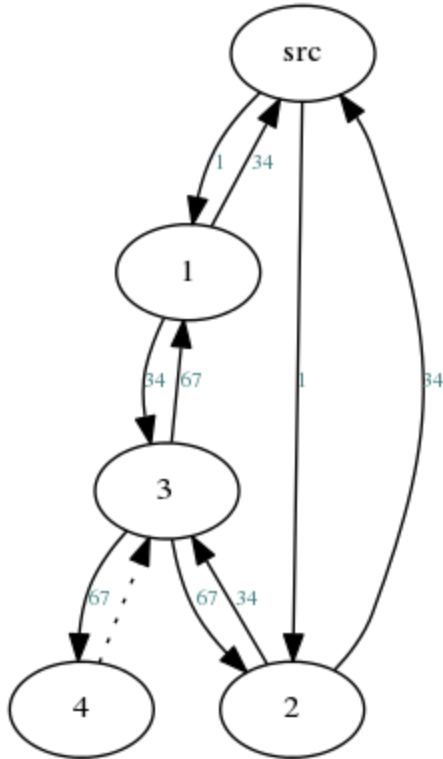
4. Output example

Output given by the program for example input data from previous section:

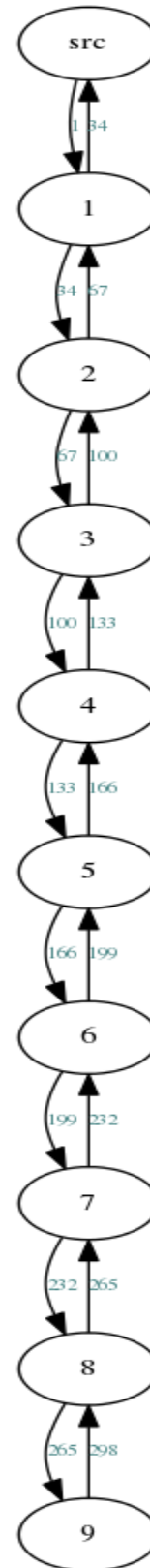
```
digraph G {
    node [fontsize=12]
    edge [fontcolor="0.5 0.5 0.5",fontsize=8]
    0 [label="src"]

    1 -> 0 [label=34]
    2 -> 0 [label=34]
    0 -> 1 [label=1]
    3 -> 1 [label=67]
    0 -> 2 [label=1]
    3 -> 2 [label=67]
    1 -> 3 [label=34]
    2 -> 3 [label=34]
    4 -> 3 [style=dotted]
    3 -> 4 [label=67]
}
```

Visualization of above output using graphviz:

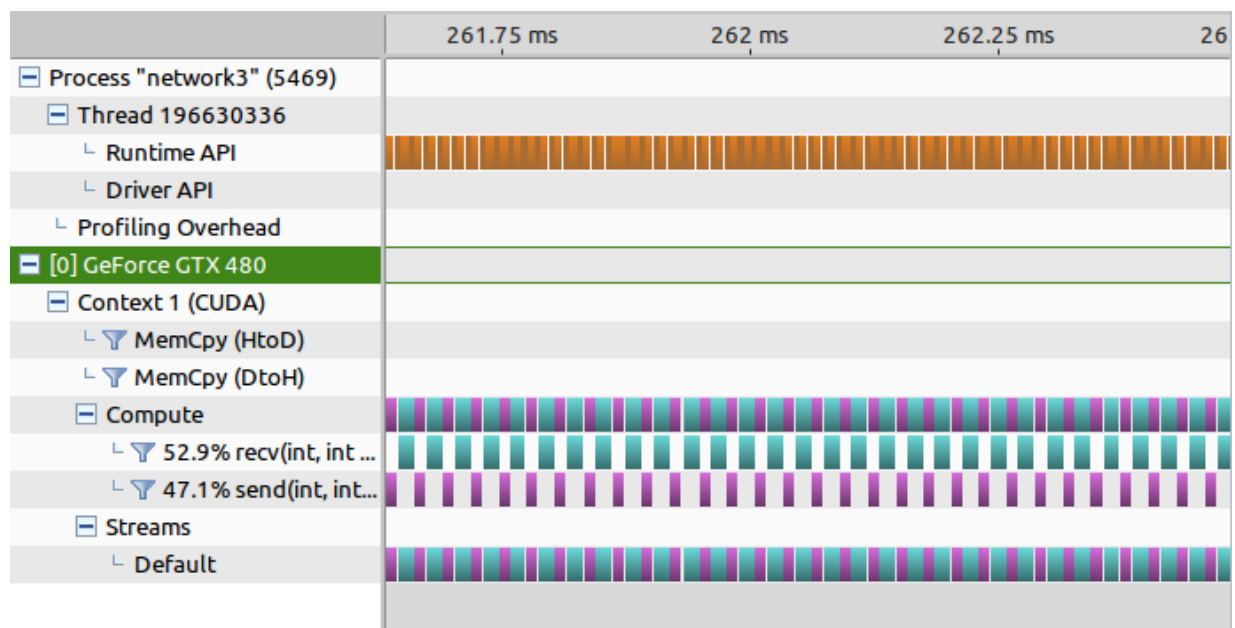


Another visualization:



5. Versions of program and optimizations

- v1. One kernel with `__syncthreads` between receiving and sending stages, only one block.
- v2. Many executions of simple kernels without loops, many blocks, iterating over edges, using atomic operations, improved memory allocation, e.g. two arrays of size equals to the number of edges are used in kernel but only one in host, so host is allocating only one array.
- v3. Many blocks, iterating over nodes, kernels with loops.
- v4. Decreased number of memory operations in receive kernel by copy params from global memory to local variables; using counter (protected by atomic add) in order to check if any changes have been made between iterations so the simulation can terminate sooner.



Timeline from cuda profiler in Nsight IDE (profiling v3).

i Memory Bandwidth And Utilization

The following table shows the memory bandwidth used by this kernel for the various types of memory on the device. The table also shows the utilization of each memory type relative to the maximum throughput supported by the memory. [More...](#)


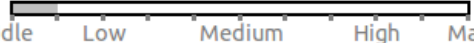
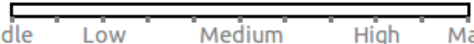
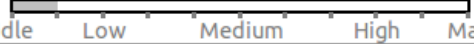
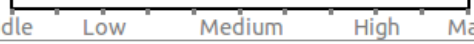
	Transactions	Bandwidth	Utilization
L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	359	5.335 GB/s	
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Global Loads	1394	20.714 GB/s	
Global Stores	292	2.667 GB/s	
L1/Shared Total	2045	28.716 GB/s	
L2 Cache			
Reads	1648	6.122 GB/s	
Writes	774	2.875 GB/s	
Total	2422	8.997 GB/s	
Texture Cache			
Reads	0	0 B/s	
Device Memory			
Reads	176	653.819 MB/s	
Writes	770	2.86 GB/s	
Total	946	3.514 GB/s	
System Memory [PCIe configuration: Gen2 x16, 5 Gbit/s]			
Reads	0	0 B/s	
Writes	0	0 B/s	
Total	0	0 B/s	

Memory bandwidth report from cuda profiler (profiling v3).

i Memory Bandwidth And Utilization

The following table shows the memory bandwidth used by this kernel for the various types of memory on the device. The table also shows the utilization of each memory type relative to the maximum throughput supported by the memory.

[More...](#)

	Transactions	Bandwidth	Utilization
L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	360	5.556 GB/s	
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Global Loads	940	14.507 GB/s	
Global Stores	150	2.18 GB/s	
L1/Shared Total	1450	22.243 GB/s	 Idle Low Medium High Max
L2 Cache			
Reads	1248	4.815 GB/s	
Writes	605	2.334 GB/s	
Total	1853	7.149 GB/s	 Idle Low Medium High Max
Texture Cache			
Reads	0	0 B/s	 Idle Low Medium High Max
Device Memory			
Reads	176	679.045 MB/s	
Writes	693	2.674 GB/s	
Total	869	3.353 GB/s	 Idle Low Medium High Max
System Memory [PCIe configuration: Gen2 x16, 5 Gbit/s]			
Reads	0	0 B/s	
Writes	0	0 B/s	
Total	0	0 B/s	 Idle Low Medium High Max

Memory bandwidth report from cuda profiler (profiling v4).
One can see the better results (global loads/stores, L1, L2) compared to v3.

6. Tests

Available devices:

- GeForce GTS 450
- GeForce GTX 480

Each presented execution time is an average based on 5 runs in ms.

For clique $N=10000$ and higher there are no results due to error: "the launch timed out and was terminated in ...".

Test cases:

name	type (description)	number of vertices	number of edges*
10a	clique (dense graph)	10	45
100a	clique (dense graph)	100	4950
1000a	clique (dense graph)	1000	499500
10b	path (sparse graph)	10	9
100b	path (sparse graph)	100	99
1000b	path (sparse graph)	1000	999
10000b	path (sparse graph)	10000	9999
100000b	path (sparse graph)	100000	99999
1000000b	path (sparse graph)	1000000	999999

*Given graph $G(V, E)$ is undirected hence a graph representation in the program has twice as much edges.

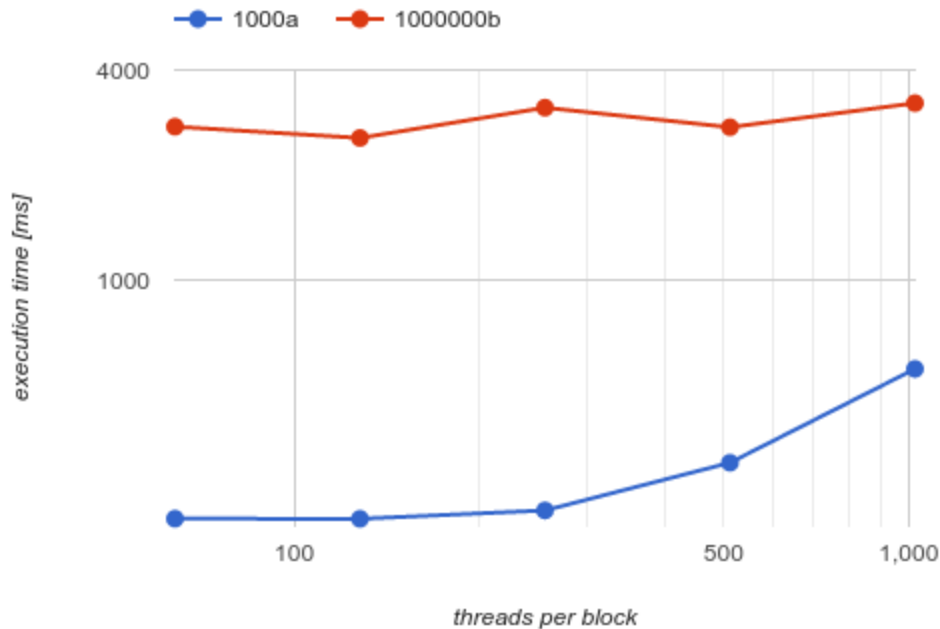
Test A. Execution time vs number of threads per block

Conditions:

graphic card	GeForce GTX 480
version	v4

Results:

threads per block test	64	128	256	512	1024
1000a	209.35	209.16	221.01	302.42	560.47
1000000b	2755.45	2558.18	3122.04	2745.3	3216.31



Comment:

The best results are achieved using 128 threads per block for both test types.

Generally in tests with small number of vertices one can see that less threads per block is better. It's caused by greater number of blocks. It's recommended by CUDA Programming Guide that a program should have many blocks because it can be much better parallelized.

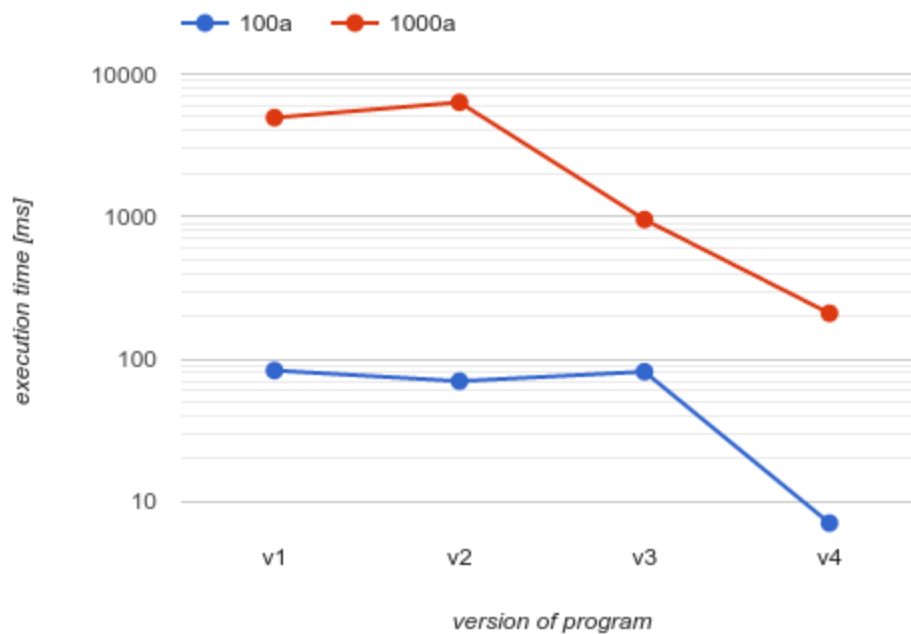
Test B. Execution time vs optimizations

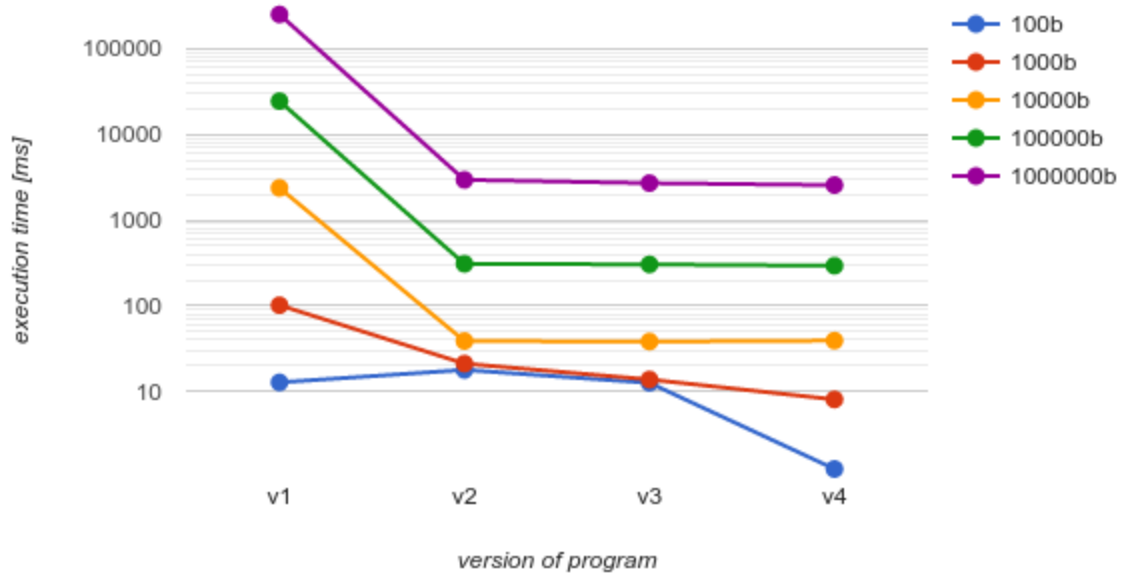
Conditions:

graphic card	GeForce GTX 480
threads per block	128

Results:

test / version	v1	v2	v3	v4
100a	83.02	69.86	81.42	7.05
1000a	4938.09	6322.82	951.57	209.23
test / version	v1	v2	v3	v4
100b	12.79	17.89	12.63	1.25
1000b	101.59	21.17	13.81	8.09
10000b	2371.63	38.83	38.31	39.25
100000b	24364.3	308.64	303.78	293.52
1000000b	250239	2942.93	2696.44	2557.75





Speedup:

test / version	v1	v2	v3	v4
100a	1	1.188376754	1.019651191	11.77588652
1000a	1	0.7809948725	5.189413285	23.60125221
test / version	v1	v2	v3	v4
100b	1	0.7149245388	1.01266825	10.232
1000b	1	4.798771847	7.356263577	12.55747837
10000b	1	61.07725985	61.90629079	60.42369427
100000b	1	78.94083722	80.20376588	83.00729081
1000000b	1	85.03056478	92.80347421	97.83559769

Comment:

The best results are achieved by v4 version of program. This version has the greatest speedups.

Program v2 was written for dense graph but it runs much slower. Probable causes of this situation are: synchronization between kernels execution, much more number of stores to global memory and frequently use of atomic operations.

Changing approach of handling data to processing nodes instead of edges in v3 allows to achieve significant speedup.

Further optimizations implemented in v4 gives the final best result, for instance speedup up to 23.60 for dense graph and speedup up to 97.84 for sparse graphs.

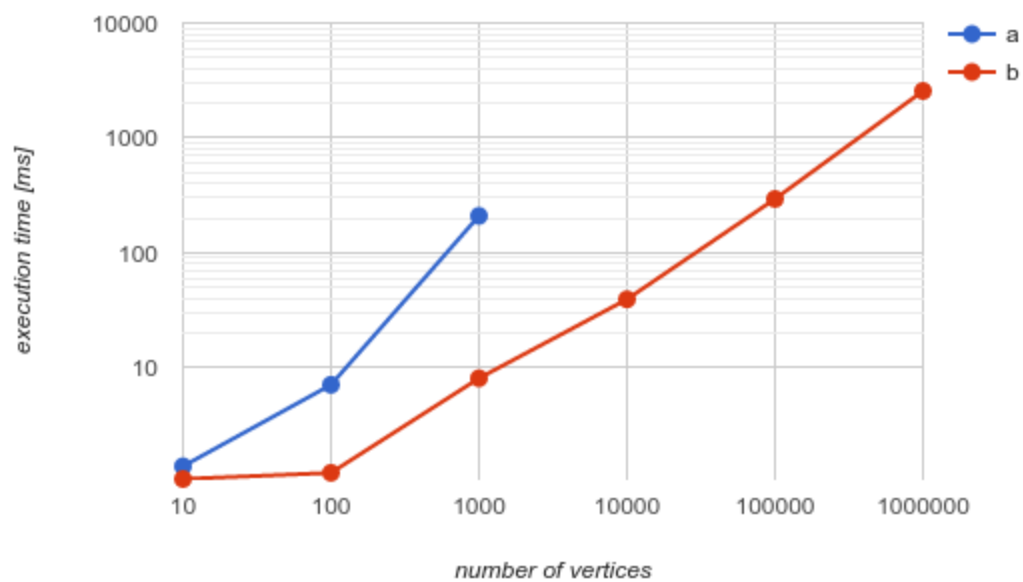
Test C. Execution time vs size of input data (number of vertices and edges)

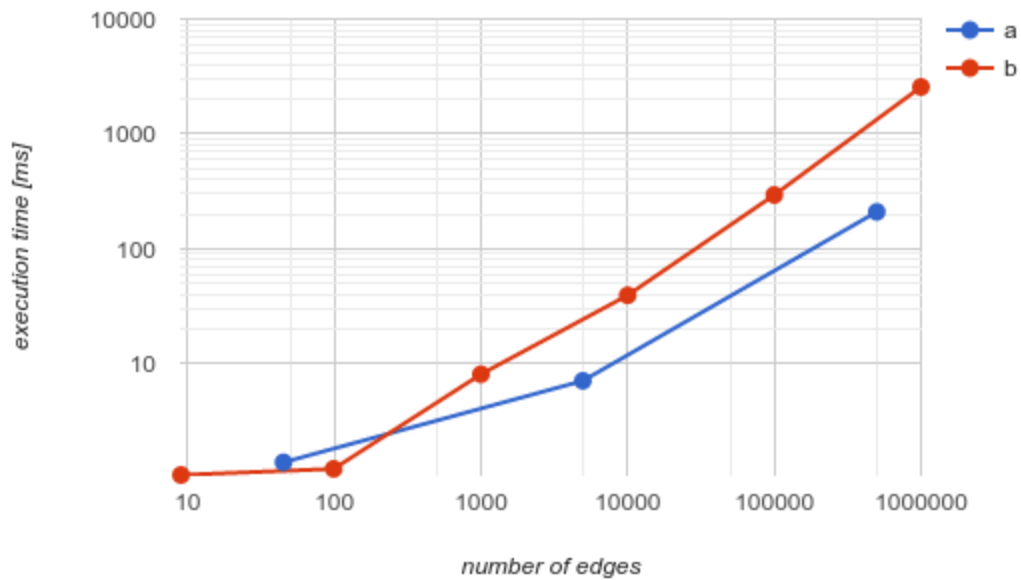
Conditions:

graphic card	GeForce GTX 480
threads per block	128
version	v4

Results:

input size	10	100	1000	10000	100000	1000000
test						
a	1.38	7.07	209.18	-	-	-
b	1.07	1.21	8.06	39.27	293.52	2558.18





Comment:

Each iteration of the algorithm should last proportionally to the $O(|V| + |E|)$.

Obtained results confirm these expectations: with the increase in the input size execution time grows linearly.

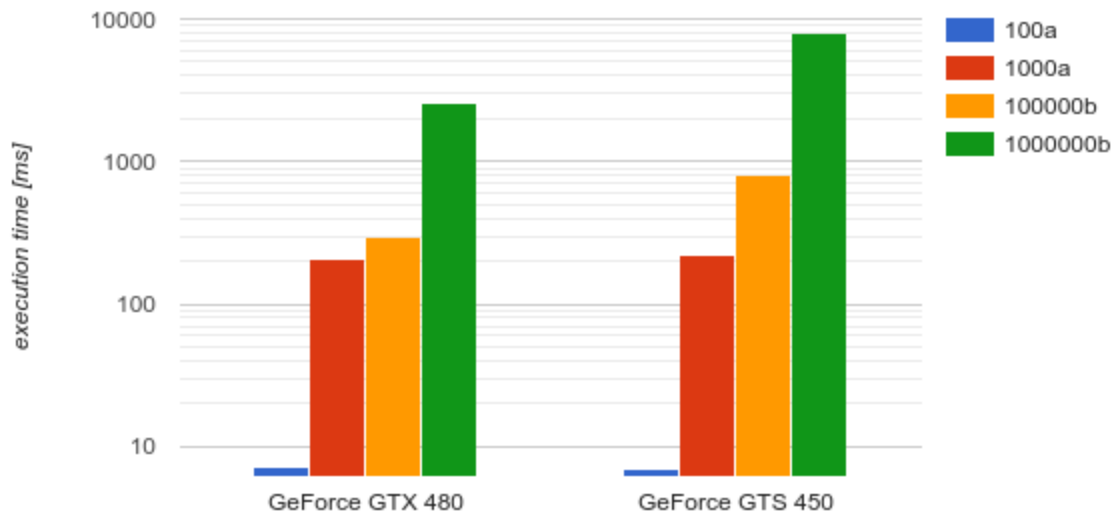
Test D. Execution time vs device

Conditions:

threads per block	128
version	v4

Results:

test / device	GeForce GTX 480	GeForce GTS 450
100a	7.07	7.01
1000a	209.13	223.69
100000b	293.39	812.7
1000000b	2557.54	7918.98



	GTX 480	GTS 450
Compute Capability	2.0	2.1
Threads per Block	1024	1024
Shared Memory per Block [KiB]	48	48
Multiprocessors	15	4
Multiprocessor Clock Rate [MHz]	1401	1622
Global Memory Size [GiB]	1.5 GDDR5	1.0 GDDR5
L2 Cache Size [KiB]	768	256

Comment:

Newer and more powerful device has better results as expected. The difference grows with the number of vertices and edges.

Chart presents that execution time for dense graphs is nearly the same for both devices whereas for sparse graphs there is a significant difference, about 3 times in favor of GTX 480.

Test E. Execution time vs size of input data, number of threads per block and optimizations

Conditions:

graphic card	GeForce GTX 480
--------------	-----------------

Results:

threads per block		64	128	256	512	1024
version	test					
1	100a	116.86	82.99	83.03	83.71	86.12
	1000a	8568.54	4938	3955.86	5161.42	5605.31
	100b	15.87	13.21	13.11	13.26	15.23
	1000b	191.75	101.56	61.72	34.96	28.43
	10000b	4482.1	2371.66	1613.33	1532.34	1436.38
	100000b	46955	24364.4	17459.6	19605.9	21885
	1000000b	464637	250238	181165	198162	221516
4	100a	6.96	7.05	7.13	7.23	7.52
	1000a	209.4	209.2	220.95	302.41	560.28
	100b	1.25	1.21	1.25	1.34	1.57
	1000b	8.1	8.1	8.84	10.31	13.17
	10000b	39.76	39.21	39.87	48.02	50.79
	100000b	300.65	293.46	357.23	308.09	361.48
	1000000b	2755.71	2557.97	3123.29	2768.46	3216.42

Comment:

The lowest execution time for version v1 is for 256 threads per block whereas 128 threads per block is better choice for version v4.

One can notice that maximum speedup for dense graphs is about 41 and for sparse graphs is about 168. In case of the lowest execution time, so when threads per blocks equals to 128, maximum speedup for dense graphs is about 24 and for sparse graphs is about 98.

With the increase in the input size execution time grows linearly as expected.

Test F. Execution time vs size of input data, number of threads per block and device

Conditions:

version	v4
---------	----

Results:

threads per block		64	128	256	512	1024
device	test					
GeForce GTX 480	100a	7.03	7.05	7.11	7.24	7.53
	1000a	209.5	209.37	220.37	302.37	560.3
	100b	1.28	1.27	1.29	1.32	1.58
	1000b	8.01	8.12	8.8	10.27	13.19
	10000b	39.91	39.34	39.96	47.99	50.93
	100000b	300.94	293.53	357.43	304.83	361.63
	1000000b	2757.11	2560.48	3124.38	2726.88	3218.7
GeForce GTS 450	100a	6.99	7.01	7.05	7.13	7.33
	1000a	224.43	223.68	224.76	287.45	528.12
	100b	1.27	1.26	1.29	1.33	1.55
	1000b	8.59	8.55	8.66	9.75	11.95
	10000b	106.56	93.92	96.45	101.18	117.6
	100000b	970.55	812.54	840.98	845.46	1067.61
	1000000b	9585.49	7918.1	8144.51	8199.2	10508.9

Comment:

Both graphic cards show similar behaviour in the context of the number of threads per block. The best results are obtained using 128 threads per block.

GTX 480 is slightly better than GTS 450 for dense graphs up to 256 threads per block, however GTS 450 is faster using 512 and 1024 threads per block.

GTX 480 is about 3 times faster than GTS 450 for sparse graphs.