# CUDA implementation of the algorithm for simulating the epidemic spreading over large networks

**Conference Paper** · January 2012

CITATIONS
0

READS
672

**2 authors**, including:

Mile Sikic
University of Zagreb
**113** PUBLICATIONS **6,094** CITATIONS

SEE PROFILE

# CUDA implementation of the algorithm for simulating the epidemic spreading over large networks

Matija Šošić*, Mile Šikić **, ***
* Faculty of Electrical Engineering and Computing, Zagreb, Croatia
** Faculty of Electrical Engineering and Computing/Department of Electronic Systems and Information Processing, Zagreb, Croatia
***Bioinformatics Institute, A*STAR, Singapore
{ matija.sosic@fer.hr, mile.sikic@fer.hr }

**Abstract – For some years now, there has been an increasing interest in modeling and analyzing the spread of epidemics in both human and computer networks. The obvious advantage a computer simulation of the epidemic spread offers is that the answer is delivered in short time and the number of hosts included in simulation can approach their real-world number. This paper presents a CUDA (*Compute Unified Device Architecture*) technology based implementation of the simulation algorithm for modeling of the epidemic spread on a network. Spreading of the epidemics over the network is modeled using discrete SIR (*Susceptible - Infected - Recovered*) model. This implementation offers selection of a starting node and monitoring of the epidemic spread in each cycle. Compared to a common CPU implementation, the CUDA version achieves about 10x faster execution time in the worst case. That speed up is of great significance when running tests on large networks. The implementation was tested on real social networks consisting of more than 5 million nodes. Hence, we believe it can be of a practical value in analysis of the epidemic spreading over large networks. To the best of our knowledge, this is only implementation of SIR model on CUDA.**

## I. INTRODUCTION

Understanding the spread of epidemic in populations is a key to controlling them. Computational simulations of epidemics provide a valuable tool for the study of dynamics of epidemics. In such simulations, populations are represented by networks, where hosts and their interactions among each other are represented by nodes and links. SIR *(Susceptible – Infected – Recovered)* compartment model is used for monitoring epidemic spread in network where each host in a population can belong to the one of three compartments. Those compartments are Susceptible (consisting of hosts who can get infected), Infected (consisting of hosts who are infected and can infect others) and Recovered (holding hosts who have recovered and are immune). For realistic scenarios it is necessary to simulate on large numbers of nodes. When dealing with extremely large networks it is crucial to achieve a reasonable execution time of a conducted simulation. In public health studies, realistic epidemic simulations (EpiFast [1], EpiSims [2] and EpiSindemics [3]) that include a lot of parameters are widely used. Basic underlying simulation algorithms Naive SIR and FastSIR are described in [5]. Since these algorithms are CPU implementations, in this study we are trying to outperform them using parallelization of the Naïve SIR algorithm using CUDA architecture.

CUDA *(Compute Unified Device Architecture)* is a parallel computing architecture developed by Nvidia. It is a computing engine in Nvidia GPUs *(Graphics Processing Units)* that is accessible to software developers through different programming languages. Through CUDA developers get access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs, making them accessible for computation just like CPUs. Unlike CPUs, however, GPUs have a parallel throughput architecture that emphasizes executing many concurrent threads slowly, rather than executing a single thread very quickly. With a certain range of problems, this becomes a more natural and more time-efficient way of implementing a solution.

We define a network as an undirected and non-weighted graph G(*N*, *L*) (*N* -set of nodes, *L* -set of links). Link (*u*, *v*) exists only if the two nodes *u* and *v* are in contact during epidemic time. We also assume that the network is static during epidemic process, meaning that the set of links *L* does not change during simulation. In other words, links cannot appear or disappear. To simulate the epidemic propagation through the network, we used the discrete stochastic SIR model. The initial conditions that will determine the course of simulation are start node and the epidemic parameters *p* and *q*. Parameter *p* is a probability that the infected node *u* infects adjacent susceptible node *v* in one discrete time step. Parameter *q* is a probability that the infected node recovers in one discrete time step. In this model each node at the same time can be in only one of the following states: susceptible (S), infected(I) and recovered(R). When there are no more infected nodes simulation is over.

Section Methods describes the general idea behind implementation. In section Results we present the speedup achieved over Naive SIR and FastSIR on a specific test case. The last section (Discussion and conclusion)

**CUDA SIR algorithm** - CPU wrapper

**Input:** *(G, Nodes, p, q)* where G is contact network, *Nodes* is a structure holding information about current state of each node. *I(Nodes)* returns *true* if there is at least one infected node. *N(Nodes)* returns *true* if there is at least one susceptible node whose neighbor is infected.
**Output:** Updated structure *Nodes* - for each node holds information in which step it was infected.

```
while I(Nodes) and N(Nodes) do
  Nodes = expandSISD(Nodes)
end while

if not I(Nodes) then
  while I(Nodes) do
    Nodes = recoverOnly(Nodes)
  end while
end if
output Nodes
```

**CUDA SIR algorithm** - GPU function **expandSIMD**

**Input:** *(G, Nodes, p, u, warpSize)* . *S(v)* returns *true* if node *v* is susceptible. *u* is concrete node upon which is this function called.
**Output:** Updated structure *Nodes.*

```
determine offset - from which neighbor to start
for each warpSize - th neighbor v of u (starting from offset) do
  if S(v) is equal to true then
    let transmission of infection from u to v occur with probability p
    if infection does occur then
      update S(v) and Nodes
    end if
  end if
output Nodes
```

**CUDA SIR algorithm** - GPU function **expandSISD**

**Input:** *(G, Nodes, q)* . *I(v)* returns *true* if node *v* is infected.
**Output:** Updated structure *Nodes*

```
NodeGroup = subset of Nodes to inspect
for each node v of NodeGroup do
  if I(v) is equal to true then
    Nodes = expandSIMD(Nodes, v)
  end if
  let recovery of v occur with probability q
  if recovery does occur then
    update I(v) and Nodes
  end if
end for
output Nodes
```

Figure 1. CUDA SIR pseudo code

proposes some ideas for eventual further research related to this paper.

## II. METHODS

At the beginning of the epidemic simulation all nodes in the graph G are in the susceptible state, except for the arbitrarily chosen starting node. It is initially infected and it attempts to transfer the infection to its neighbors with probability *p*. Infected nodes in the next step attempt to transfer the infection to their neighbors. After finishing its infection step, each node attempts to recover with probability *q*. Simulation continues until each node is either susceptible or recovered.

The Naive SIR [5] algorithm sequentially loops through the infected nodes and then attempts to infect their adjacent nodes by sequential looping through a list of their neighbors. Infected nodes which have to be processed in a current step maintain a *frontier*. The Naive SIR algorithm uses a queue *as a frontier*. When a node gets infected, it is assigned a number of current time step and is added to the end of the queue. A node can leave the queue when it successfully recovers. The simulation is over when the *frontier* is empty. We notice that this algorithm exposes its parallel nature on two levels: it is possible to process many infected nodes simultaneously and for each of infected nodes the neighbors can also get infected simultaneously. If information of the step in which the node changes state is preserved, the parallel nature of the executing algorithm will not affect the result in any way.

The FastSIR algorithm [5] uses probability distributions of the number of infected nodes to speed up recovery time of infected nodes to one discrete step during a simulation of the epidemic spreading to reduce running time. The speed up of the FastSIR algorithm compared with the Naive SIR algorithm is proportional to the average value of the infectious period. However, the FastSIR algorithm does not follow the epidemic dynamics. It only provides information of the final number of infected nodes during simulation.

The described simulation problem resembles graph traversal problem [6], but in this case it is not necessary that each node gets "visited". For example, with a high value of *q* and low value of *p*, an epidemic stops spreading shortly after start and most of the nodes will never leave the susceptible state. The simplest case is when *p* and *q* are set to 1. This simplifies the described problem to the well known BFS (*Breadth First Search*) as the each node gets visited exactly once. In general case where *p* and *q* are arbitrary values from the interval [0, 1] the algorithm resembles BFS, but with different propagation conditions. That is why implementing BFS on CUDA is a great step to final algorithm for epidemics spreading. GPUs have shown promising results in accelerating computationally challenging network problems but their performance depends heavily on the structure of the network. When a network has a highly irregular structure, as most real-world networks tend to have, it can significantly slow down the program execution. CUDA BFS [7] algorithm, upon which we built our solution, addresses this problem quite successfully. The main idea is to divide nodes into groups of equal size. Then, to each node group is assigned a group of threads. In CUDA terminology, group of threads is usually called a *warp*. Each thread from *warp* sequentially inspects nodes from its node group. If inspected node is infected, each thread from a *warp* "attacks" one of its neighbors (or more, if there are less threads in a *warp* than there are nodes in a node group), trying to infect it. As multiple *warps* are executed at the

same time on different processing units of CUDA GPU, two-level parallelization is achieved. First level of parallelization is processing multiple infected nodes at the same time. The second level is parallel processing (attempting to infect them) of neighbors or the infected node. Although it may seem more efficient to assign *warp* to single node and evade sequential „walk“ through a group of nodes, in real CUDA system it would result with too many inactive threads causing underutilization and therefore slowing the performance.

As the *frontier* is a thread-shared resource, it would be inconvenient to implement it as a queue. Such a structure would require thread-safe operations of adding/taking element. That would result in serializing all queries towards it as only one thread can access it at a time in order to preserve correctness of the algorithm. In our solution we do not explicitly maintain a *frontier*, but store the information about the current state of each node. For every node it is enough to store in which step they got infected and whether they are currently immune. We eliminate the need for storing information on the node's susceptibility by simple reasoning: if a node is not infected, but also is not immune, then it is susceptible. Internally, we store this information in two arrays of size $|N|$ - *Levels* and *Immune,* while overall structure is named *Nodes* (used in Figure 1). As a consequence, *frontier* does not exist independently but is scattered in *Nodes.* Because of that, in each step it is necessary to loop through all the nodes and process only those marked as infected.

In this implementation the network is represented as an adjacency list. Since we always access all of the neighbors, we do not need to know whether two specific nodes are linked. Further optimization of this representation is achieved by packing rows of adjacency list into a single large array making it more convenient to store and use on GPU. In this way, a network is being held in two arrays, $N$ and $L$. Figure 2 visualizes this data structure. Array $L$ holds rows of adjacency list merged sequentially. At $N[i]$ is stored position in $L$ where first neighbor of $i$-th node is stored. The described data structure is also known as compressed sparse row (CSR) in sparse – matrix computation domain [4].

Before starting the simulation part of the algorithm, random number generator (RNG) has to be initialized for the each thread. RNGs are used to simulate probability of occurrence of an infection transfer and recovery of a node, represented by $p$ and $q$ respectively. CUDA provides its own library for generating random numbers and it is recommended that each thread uses its own generator to evade possible determinism. The following two procedures, loading of the network from a file and initializing an array of RNGs, have to be executed every time the program is started. Time needed for this depends on the size of the network and the CPU used in system. In our test case for LiveJournal network [8] it took about 20 seconds. Once network is loaded into RAM and RNGs are initialized, an arbitrary number of independent simulations can be executed, so the amount of time spent for those two steps can usually be ignored due to large number of conducted simulations.

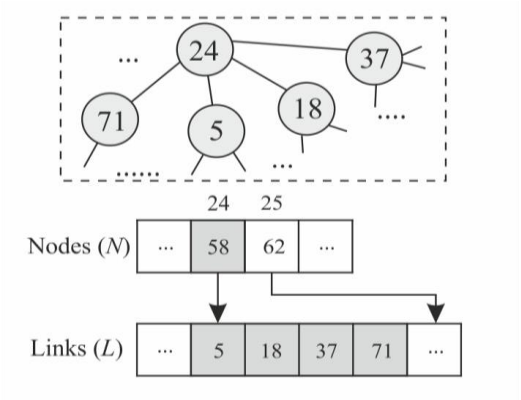


Figure 2. Graph data structure used in in the implementation

The simulation is executed in steps, where for each of the steps a suitable CUDA function is called. Between every two steps of the simulation, CPU analyses information about the current state in the network and decides which CUDA function has to be called next. Each node is assigned a certain level, presenting in which step it was infected. Node levels are stored in the array *Levels*. If the node is not infected, its level is considered to be infinite. Node that is chosen to be the source of infection has level of 0. Figure 3 shows an example of the first three steps in the epidemic spread, presuming $p$ is 1. The brightest node with level 0 is the source of epidemic. In the first GPU call the *frontier* consists only of this node. As $p$ is 1 in this example, four nodes are infected in the first step and thus assigned level of 1. Algorithm continues in the similar manner producing final state of the network shown in Figure 3.

CUDA function which operates on GPU alternates between two phases: SISD (*Single Instruction Single Data*) and SIMD (*Single Instruction Multiple Data*) phase, as it is shown in Figure 4. In the SISD phase all threads in a *warp* execute the same instruction on the same data assuring there is no divergence slowing the performance. Moreover, in this phase each thread group copies its portion of the data from the slow global to the fast shared memory. Accessing global memory in CUDA architecture takes around 300 cycles while access to shared memory takes only few [8]. In SISD phase all threads in a *warp* loop sequentially through assigned nodes. If the currently inspected node is infected, SIMD phase is applied to its neighbors. In SIMD phase the threads from the same *warp* execute the same instructions but on the different data. In this case array holding the node neighbors is being processed simultaneously by the whole *warp*. Thread divergence is present in SIMD phase, but due to the nature of the problem it cannot be evaded. Figure 4 depicts cooperation of SIMD and SISD phases more clearly.

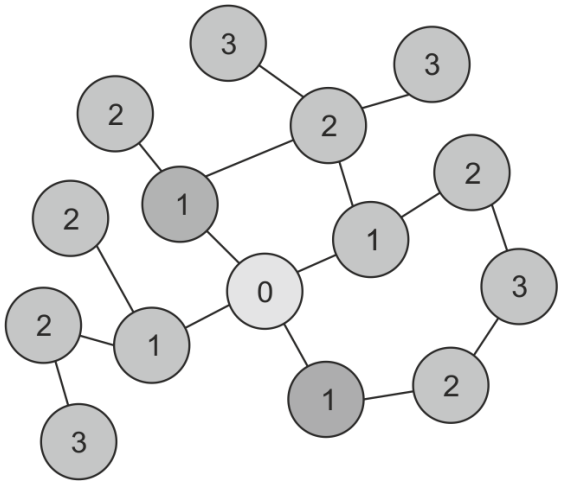


Figure 3. Steps in epidemic spread, the node marked with 0 is source, p equals 1

Example in Figure 4 presumes size of *warp* to be 3 and number of nodes assigned to it to be 6. Only the fourth node is infected. All three threads are in SISD phase until they reach the infected node, when SIMD phase is initiated. Threads split and each one tries to infect different neighbor. After all neighbors of infected node have been processed (tried to get infected), the algorithm returns to SISD phase. Another problem is determining optimal sizes for a *warp* and assigned group of nodes. *Warp* size is partially determined by certain architectural traits of underlying CUDA system (should be multiplier of 8), but both parameters mostly depend on structure of the given network. So far, we have determined optimal parameters purely experimentally, but it would be interesting to see if they can be predicted from input network characteristics.

In a case when *q* is very small and *p* high, network quickly comes to a state where the infected nodes do not have any susceptible neighbors. In that particular case it is redundant to spend time on visiting neighbors of the infected node just to find out that none are susceptible. When this state occurs in the network algorithm recognizes it and changes its behavior by calling another CUDA function, which then assigns only one thread to each of the infected nodes. In Figure 1 it is achieved by recoverOnly function.

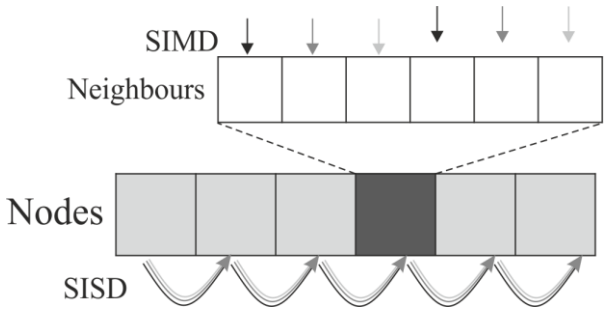


Figure 4. SISD and SIMD phases alternate during algorithm execution

When there are no more infected nodes left, the simulation is finished. It is possible to reconstruct the epidemics spread from array *Levels* which holds the moment of the infection for each node.

## III. RESULTS

Purpose of this section is to show the execution time in comparison to Naive SIR and FastSIR algorithms. Tests were conducted on server with Nvidia 570 GTX GPU and Intel Core2 6400@2.13 GHz with 4 GB RAM.

Tests were done on a real network example of a social network LiveJournal. LiveJournal is a free online community with almost 10 million members. LiveJournal allows members to maintain journals and to declare other members as friends. For the following test case a network representing friendship relations of this population (nodes represent users and links represent friendships) is used. Results for the Live Journal network [8] consisting of 5 million nodes and 50 million links are shown in Table 1.

The running time ratio for FastSIR and CUDA algorithm is especially interesting. The achieved speedup ranges from 5 to almost 30 times in some cases. On average, CUDA algorithm is about 16 times faster than FastSIR for the LiveJournal network. Figure 4 shows the calculated ratios for the different choices of *p* and *q*.

TABLE I. RUNNING TIME IN SECONDS FOR 2000 SIMULATIONS FOR LIVE JOURNAL NETWORK

| q | p = 0.2 | | | p = 0.5 | | | p = 0.8 | | |
|---|---|---|---|---|---|---|---|---|---|
| | Naive SIR | FastSir | CUDA | Naive SIR | FastSir | CUDA | Naive SIR | FastSir | CUDA |
| 0.1 | 50683 | 6699 | 1160 | 48373 | 5635 | 960 | 47531 | 5078 | 820 |
| 0.2 | 25841 | 7200 | 700 | 24398 | 6314 | 620 | 24067 | 5357 | 560 |
| 0.3 | 18550 | 7200 | 500 | 16580 | 6841 | 460 | 16276 | 5609 | 420 |
| 0.4 | 13686 | 6987 | 440 | 12870 | 7259 | 380 | 12329 | 5843 | 360 |
| 0.5 | 13197 | 6704 | 380 | 10394 | 7591 | 360 | 9951 | 6060 | 320 |
| 0.6 | 9394 | 6400 | 320 | 8720 | 7859 | 320 | 8345 | 6253 | 280 |
| 0.7 | 8301 | 6073 | 220 | 7513 | 8072 | 280 | 7185 | 6429 | 280 |
| 0.8 | 8744 | 5805 | 280 | 6622 | 8250 | 260 | 6293 | 6592 | 240 |
| 0.9 | 7521 | 5508 | 200 | 5869 | 8555 | 180 | 5597 | 6749 | 240 |
| 1 | 5291 | 5259 | 180 | 5064 | 8666 | 180 | 5082 | 7777 | 140 |

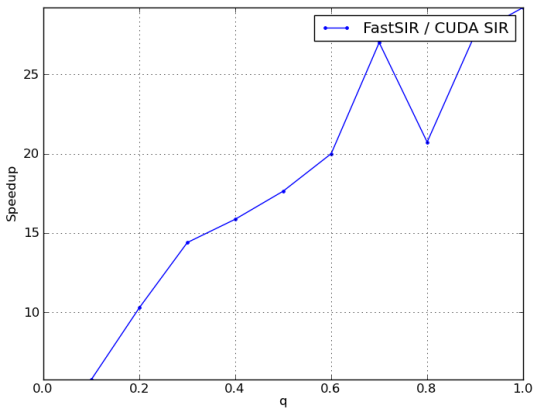


Figure 5. Running time ratio for FastSIR and CUDA algorithm. Parameters: 2000 simulations, p = 0.2, q = 0.1 to 1

As it can be seen from Table 1 and Figure 5, the speedup gradually increases with the growth of the epidemic parameter *q*. It is important to notice that unlike CUDA SIR and Naïve SIR, the FastSIR algorithm does not follow epidemic dynamics in time. If we make a comparison with Naive SIR, we can see from Table 1 that CUDA SIR can be faster more than a few hundred folds.

## IV. Discussion and Conclusion

In this paper we presented a general idea behind the parallel implementation of the SIR algorithm for simulating epidemic spread in networks. We have shown that it is possible to achieve significant speedups up to 30 folds and a few hundred folds compared to FastSIR and Naive SIR respectively. This becomes more evident as the size of networks grows. In the future work it would be interesting to see a visualization tool for monitoring the progress of the simulation step by step. Additional speedup could be achieved by using multiple GPUs for the calculations and the MPI could be used for running more simulations at the same time. Current SIR model could be broadened by adding new events to the network, like disappearing link or dynamical changes of *p* and *q* in order to represent population and epidemic conditions more realistically.

[1] G. K.R. Bisset, J. Chen, X. Feng, V.A. Kumar,M.V.Marathe, Epifast: a fast algorithm for large scale realistic epidemic simulations on distributed memory systems, in: Proceedings of the 23rd international conference on Supercomputing, ICS '09, ACM, New York, NY, USA, 2009, pp. 430–439.

[2] C.L. Barrett, K.R. Bisset, S.G. Eubank, X. Feng, M.V. Marathe, EpiSimdemics: an efficient algorithm for simulating the spread of infectious disease over large realistic social networks, in: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, SC '08, IEEE Press, Piscataway, NJ, USA, 2008.

[3] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, J. Wiener, Graph structure in the Web, Computer Networks 33 (2000) 309–320.

[4] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. In *Proc. Conf. Supercomputing (SC'09)*.

[5] Nino Antulov-Fantulin, Alen Lancic, Mile Sikic, FastSIR Algorithm: A Fast Algorithm for simulation of epidemic spread in large networks by using SIR compartment model, arXiv:1202.1639v1[cs.DS]

[6] T.H.Cormen, C.E.Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, The MIT Press, New York, 2011.

[7] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, Kunle Olukotun, Accelerating CUDA Graph Algorithms at Maximum Warp

[8] Stanford large network dataset collection, http://snap.stanford.edu/data/index.html, 2009.

[9] NVIDIA CUDA C Programming Guide, version 4.0