Sequential Johnson's APSP Algorithm on GPU

Anila Batool, Muntazir Mehdi

Abstract— A quite ordinary issue while processing graphs is to find the shortest distance from one node to all the other nodes. It is called all-pairs shortest path. The applications of finding shortest paths are several including Digital Mapping, Social Networking, Telephone Networks, IP Routing, Fighting Agenda, Robotic Path and many more. Although a lot of work has already been carried out in this aspect but it has been observed that it is quite difficult to exquisitely process graphs which contain a very larger number of nodes. This paper aims to provide a pertinent solution to this problem by proposing three different versions of Johnson's shortest path solution in parallel architecture over Graphic Processing Unit which resolves APSP problem. As compared to processing extensively large graphs on CPU, proposed architecture will provide a 4.5 time efficient solution for APSP problem.

Index Terms— All-Pair Shortest Path, Compute Unified Device Architecture (CUDA), Graphical Processing Unit (GPU), Johnson's Algorithm, Parallel Processing.

1 Introduction

Finding the shortest path is one of the most important topics of graph processing. Three major solutions to APSP include Dijkstra's algorithm [1], Floyd-Warshall algorithm [2] and Johnson's Algorithm [3]. Dijkstra's algorithm finds shortest path for a single node by updating the relevant weights of the adjacent nodes. Time complexity of Dijkstra's algorithm is O (V^2) where V represents the number of elements in the graph. Another upgraded version [4] of the algorithm utilizes priority queue which as a result reduces the time complexity to O (Vlog (V)). When applied on each node separately, Dijkstra's algorithm can be used to solve APSP problem. Time complexity of applying the Dijkstra's algorithm on each node of the graph is O (V^2log (V)).

However, Dijkstra's algorithm lacks the ability to work with negative weights. On the contrary, Floyd-Warshall algorithm can be used to handle positive as well as negative weights. With time complexity O (V^3), Floyd-Warshall algorithm calculates shortest path between two nodes by comparing total number of possible paths between the nodes. The third solution is Johnson's algorithm [5] which resolves the problem by using Bellman-Ford [6] algorithm to remove negative weights then uses Dijkstra's algorithm on each separate node to compute shortest paths between all the nodes in a graph. The time complexity of Johnson's algorithm is O (VE + V^2log (V)) where V represents nodes and E represents edges in a graph.

Feasible solution to the problem cannot be obtained by sequential processing of the graphs as we can observe that these algorithms are computationally costly for graphs with very large number of nodes and vertices. To resolve this problem we suggest a parallel computing architecture which will speed the overall process of finding all-pair-shortest-path about 4.5 times. This paper will illustrate the idea of using GPUs for parallel computing which provide high computational power at a lower cost as compared to other parallel computing architectures such as CRAY supercomputers. We tend to provide three different parallel computing architectures of Johnson's algorithm on Graphic Processing Units. In the end of the paper, we will provide a comparative report which will compare the three proposed solutions in terms of performance and will also describe the pros and cons of using each.

• *Anila Batool, Department of Computer Science, Govt Degree College of Special Education, Sargodha, Pakistan. E-mail: syedanaqvi.1472@gmail.com.*

• *Muntazir Mehdi, Department of Computer Science, Superior Group of Colleges, Mianwali, Pakistan. E-mail: naqviofficial5@gmail.com.*

2 Related Work

The use of GPUs for general purposes has increased dramatically in the last decade. Harish [7] used V^2 threading for parallel implementation of Floyd-Warshall. Agarwal [8] used two-flag approach over Bellman-Ford algorithm and reduced the execution time by finding the edges which should be dropped in the upcoming iteration. Meyer [9] introduced delta stepping in order to achieve parallel implementation of Dijkstra's algorithm. Bader [10] used CRAY supercomputers for parallel implementation of shortest path problem but the cost of the hardware was quite high. Singh [11] used edge-based and node-based approach and implemented modified Dijkstra's algorithm in parallel computing architecture.

We took the idea of edge classification and queue method from Busato [12]. We utilized Crauser [13] idea of parallel Dijkstra's algorithm with frontier flag and queue structure to avert branch divergence and keep track of nodes to be operated.

3 CUDA Overview

GPU provides Single Program Multiple Data Model which allows executing similar set of commands on different data items in parallel. In this technique a lot of light weight thread used by General Processing Unit are mapped into multiple cores of General Processing Unit. To implement this, our General Processing Unit should be programmed into popular framework for programming CUDA [14] that is supported by NVIDIA [15]. There are multiple Streaming Multiprocessor in the NVIDIA General Processing Unit on chip L2 cache. These Streaming Multiprocessor are consists of a lot of independent processing units. Shared Memory is a very fast and private memory this is accessible to all Streaming Multiprocessors in the system. The programmer using CUDA writes a set of commands using device kernel and the General Processing Unit executed all these commands. One block consists of multiple number of threads, and we assign this thread to single core of

General Processing Unit. Similarly a whole block of thread can also be assigned to a Streaming Multiprocessor. By using this approach each thread or even each block of thread is assigned by a unique id so that these thread and block of threads can be identified by Streaming Multiprocessor.

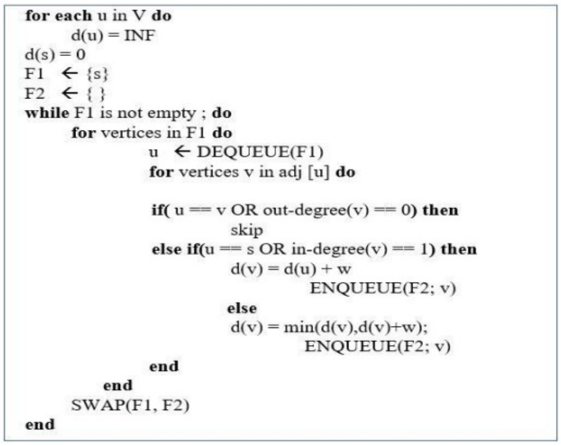
4 Sequencial Johnson's Algorithm

Johnson Algorithm is the outcome of three different phases. First, an extra node is added to the graph G and this node is called q. This node is connected to all other nodes with zero weighted edges. After this, using Bellman-Ford algorithm [6] the shortest path from this newly created node q to all other nodes of graph is evaluated. After that, the cost of all the edges in the graph is re-calculated by using the outcome of the initial phase. The last step includes the process of calculating the distance between each node present in the graph with the help of re-weighted edges obtained in the previous step.
Finally, the adjacency list is formed in which we store our graph. By using adjacency lists the graph G (E, V) will be displayed as: array is used to represent the nodes/vertices and named as Va, second array Ea is used to store the edges directly associated with the node in progress. Every new insertion in the array point to the vertex in the corresponding array. So with the help of this representation when we are trying to process the nodes in parallel we can easily access the adjacencies of a node in the graph.

5 Bellman-Ford Single Source Shortest Path Algorithm

According to Bellman's solution the cost from starting node is initialized to zero and the cost from all other nodes is initialized to infinity. In the next step, each edge is passed through a relaxing operation. Relax operation is performed equal to the total number of vertices in the graph.
Modified version of Bellman-Ford algorithm [16] includes two modifications. The modified version suggests the idea of keeping track of the vertices which are being processed by creating a separate storing structure. It is accomplished with the use of queue structure and by using lesser number of relaxing operations which results in a decrease in the execution time. The second advancement is to perform edge categorization. The edges are categorized into 4 separate classes. It results in a decrease in the number of relaxing operations. Edge classification categories are following:

- **Self-revolving edge category:** those edges whose all sides are connected to the similar nodes are categorized into self-revolving edge category. These edges are not relaxed.
- **Initial edge category:** It is the class to represent the edges connected to the source point. Relaxing operation is performed for this type by updating the value of initial nodes directly.
- **Incoming edge category:** Edges which are meant to be visited once because these have only 1 incoming edge are categorized into this category and there is no need of relaxation these edges.
- **Outgoing edge category:** The category keeps track of those edges whose have no further outgoing edge. Due to this reason, we overlook them while the algorithm is running. However, their cost is added at time when the algorithm stops.



```
for each u in V do
        d(u) = INF
d(s) = 0
F1  ← {s}
F2  ← { }
while F1 is not empty ; do
        for vertices in F1 do
                u  ← DEQUEUE(F1)
                for vertices v in adj [u] do

                if( u == v OR out-degree(v) == 0) then
                        skip
                else if(u == s OR in-degree(v) == 1) then
                        d(v) = d(u) + w
                                ENQUEUE(F2; v)
                        else
                        d(v) = min(d(v),d(v)+w);
                                ENQUEUE(F2; v)
                end
            end
        SWAP(F1, F2)
end
```

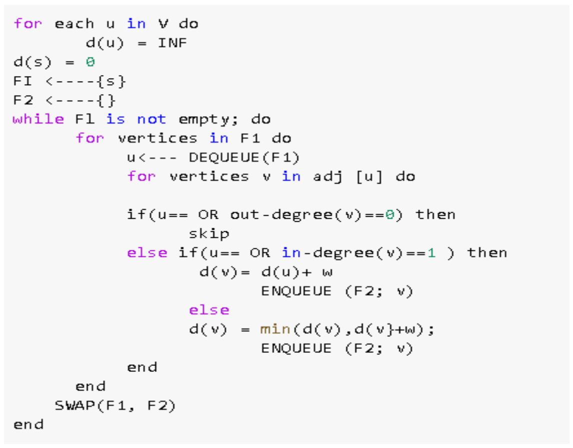
Fig. 1. Upgraded Bellman's algorithm. It shows the pseudo-code for modified Bellman-Ford algorithm.

5.1 Reweighting

The results of the first step are utilized in the process of edges' weight updating. Weight updation for an edge which starts from x and ends at y with z weight is modified as following.
$z = z + h(x) – h(y)$
The function h (.) is used to calculate the shortest path between two given vertices. In this step negative weighted graphs are converted into non-negative weighted graphs which enables us to calculate the shortest paths between all the edges present in the graph.

6 Dijkstra's Single Source Shortest Path Algorithm

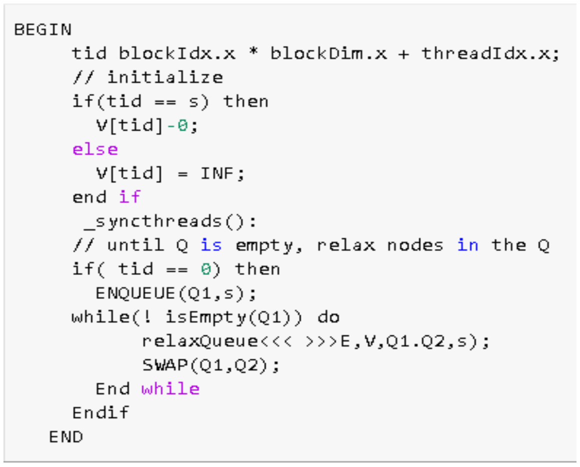
The process starts by assigning 0 distance to the source and infinity to all other. After that, two sets visited and unvisited are created. Initially it marks the source node as visited. After that, it performs relax operation for all the neighboring nodes in the unvisited set. It adds the node to the visited set as all of its neighboring nodes are done. Then new node is chosen which is not processed yet and the similar operation is repeated for that node. The algorithm completes when the unvisited set become empty. Figure 2 shows the working of generic Dijkstra's algorithm.



```
for each u in V do
      d(u) = INF
d(s) = 0
FI <----{s}
F2 <----{}
while Fl is not empty; do
     for vertices in F1 do
          u<--- DEQUEUE(F1)
          for vertices v in adj [u] do

          if(u== OR out-degree(v)==0) then
               skip
          else if(u== OR in-degree(v)==1 ) then
                d(v)= d(u)+ w
                     ENQUEUE (F2; v)
               else
               d(v) = min(d(v),d(v}+w);
                     ENQUEUE (F2; v)
          end
     end
   SWAP(F1, F2)
end
```

Fig. 2. Pseudo Code for Dijkstra's Algorithm.



```
BEGIN
     tid blockIdx.x * blockDim.x + threadIdx.x;
     // initialize
     if(tid == s) then
       V[tid]-0;
     else
       V[tid] = INF;
     end if
      _syncthreads():
     // until Q is empty, relax nodes in the Q
     if( tid == 0) then
       ENQUEUE(Q1,s);
     while(! isEmpty(Q1)) do
           relaxQueue<<< >>>E,V,Q1.Q2,s);
           SWAP(Q1,Q2);
       End while
     Endif
   END
```

Fig. 3. The Working of Bellman’s Algorithm

7 Parallel Implementation Solution

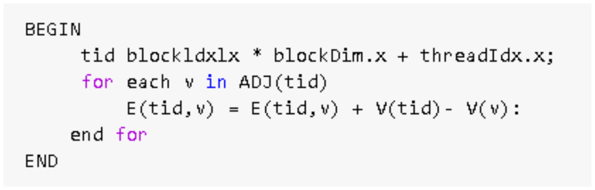
The paper presents three diverse implementations of Johnson’s algorithm. These solutions differ in the way in which to keep record of the frontier nodes at the time when Dijkstra’s algorithm is functional. First method version we propose uses a separate array to store a Boolean value which points out if a node is in frontier or not. The second method is called Q-based which puts the frontier node inside the queue with the help of atomic operation. The last method is called prefix-based which used prefix sum to put the frontier node into the queue. Initial two steps in each of these three versions are the same.

7.1 Parallel Implementation for Bellman-Ford Algorithm

Parallel Bellman-Ford algorithm makes it possible to process each single edge independent of the other edges present in the graph. A unique thread number is allocated to each thread are all the threads which are then synchronized before the next iteration. Figure 3 shows the main kernel of Bellman-Ford algorithm. As each thread starts, it processes the vertex assigned to it. The node weight array corresponding to the vertex is initialized after which all vertices except the first one are terminated. The main algorithm is controlled by the first thread. It initiates by adding the functional coding instructions inside a queue. After that, relax operation is performed for each individual node which is shown in the figure 5. The second kernel which is the child of the main kernel concurrently takes the nodes from queue. Then it relaxes the nodes extracted from the queue and puts the updated nodes in a separate queue. Then the queues are exchanged by the kernel until the time queue becomes empty. The in-queue and update operations are defined as atomic as a situation can occur in which more than one threads are trying to insert values inside the queue at the same time.

7.2 Reweighting of Bellman Ford Algorithm

Previous step enabled us to simultaneously process nodes in the kernel with the help of Maxwell architecture. As a result, each edge cost is updated independently. As we already mentioned that each nodes is carried out by a separate thread. Each thread used Bellman-Ford algorithm to calculate node weights and reassign the cost to each of the edge going out of that particular node. Figure 4 shows the formula to calculate the cost of each edge which is going away from a particular node.



```
BEGIN
     tid blockldxlx * blockDim.x + threadIdx.x;
     for each v in ADJ(tid)
         E(tid,v) = E(tid,v) + V(tid)- V(v):
    end for
END
```

Fig. 4. Reweighting Kernel

7.3 Parallel Implementation for Dijkstra’s Algorithm

This section describes the parallel implementation of Dijkstra’s algorithm. However, unlike Bellman-Ford algorithm the parallel implementation of Dijkstra’s algorithm is quite difficult due to the sequential structure of the algorithm. Various approaches can be used to parallelize the algorithm among which one is parallelization of inner operations of the algorithm. Dijkstra’s algorithm selects a node to calculate new distance values at its outer loop. The algorithm uses inner operations to update label of the nodes by relaxing out-going edges. We aim to parallelize the algorithm by providing a mechanism to choose the vertices which we process and update independently with no harm being done to the accuracy of the outcome.

At every cycle, the job is to identify the vertices which should be placed inside the group of frontier nodes. After that relaxing task can be performed for the other nodes. Crauser [13] in his paper provided an algorithm that outputs the frontier nodes with larger heuristic distances. Arranz [18] suggested a better and modified version of the algorithm provided by Crauser. In Arranz algorithm, the lowest edge cost is calculated for all outgoing edges. After that, a threshold value is calculated from the unsettled nodes processed in the first step. As a result, it enables us to put each node whose weight is larger than the threshold into frontier set. Advance reduced V3 method of CUDA is loaded in the kernel as a minimum function.

The only issue with this implementation is branch divergence. Branch divergence problem occurs when the kernel have to group up threads from various wraps but also have to branch the same target inside a new and complete tab. The method of frontier propagation is used by using queue as it is used in the Bellman-Ford algorithm to prevent branch divergence. However implementing queue in GPU is considerably different from implanting queue in CPU. In order to productively configure queue in Graphics Processing Unit, two separate tactics are used: prefix-based and Q-based.

In Q-based approach the frontier flag of the initial node is placed inside queue. The method for relaxing and updating kernel using queue based approach for parallel Dijkstra's algorithm is shown in the figure 5. The cost of modifying the algorithm in terms of flag-based approach is the cost of the total sum of atomic operations at the time when nodes are being inserted inside the queue.

```
BEGIN
    tid = blockldx.x * blockDim.x + threadIdx.x;
    nId Q[tid];
    for each v in ADJ(nId)
       if (U[v]==True) then
           V[v]= AtomicMin(V[v], V[v]+E(nld,v));
     endif
    end for
END
```

Fig. 5. Q-based parallel Dijkstra relax operations.

```
BEGIN
    tid blockIdx.x * blockDim.x + threadIdx.x;
    if (U[tid]==True and V[tid]<= A) then
        U[tid]= False
        BEGIN ATOMIC
                ENQUEUEQ.tid);
        END ATOMIC
    end
END
```

Fig. 6. Pseudo kernel for updating queue of Q-based parallel Dijkstra.

Figure 6 shows the pseudo kernel to put frontier flag of the of the source node inside a queue data structure. The use of queue in parallel Dijkstra enables us to prevent branch divergence notably. However, to prevent race conditions threads are arranged in a sequential manner while updating the queue.

7.4 Prefix Based Dijkstra's Algorithm

Another approach we used is pre-fix based which involves the addition of elements in the queue by index numbers which are calculated at the beginning of each iteration. As a result, the cost of atomic operations is eliminated at the time of filling the queue as the index numbers are already defined that are associated with each element meant to be placed inside the queue. Nevertheless, it results in another computational cost of finding of prefix sum. The kernel for updating prefix-based architecture is presented in the figure 7. While the same relax kernel which we used in the Q-based approach in figure 5 can be used in this approach as well.

```
BEGIN
      tid blockIdx.x* blockDim.x + threadIdx.x
      if (U[tid]==True and V[tid]<= A) then
  U[tid] False
         Q[ prefixSum[tid] ] = tid;
      end if
END
```

Fig. 7. Pseudo kernel for updating prefix based Dijkstra

In order to implement single source Dijkstra's algorithm for APSP problems, the same algorithm is applied to each node in the graph as a result the single source Dijkstra's algorithm can be extended to APSP parallel algorithm.

8 Experemental Setup and Result

We conducted experiments on Microsoft Windows 8 office 64-bit OS with Intel Core i5-3470 Central Processing Unit @3.2 Gigahertz and 12 GB DDR3 Random Access Memory. The Graphic Processing Unit (GPU) we used for testing is NVIDIA GeForce GTX 1050 2GB. We used 8 different graphs where each graph size is different from the other.

In this project, we made comparative analysis of four different versions of Johnson's APSP in terms of execution efficiency. Following is the list of versions which we used in this comparison report.

- Sequential Johnson's APSP
- Flag-Based version
- Queue-based version
- Prefix-based version

In addition to this, we compared speed of rate of these versions which is plotted on Figure 8.

It has been noticed that the speed-up rate of Q-based approach is faster as compared to flag-based approach for the graphs which were used in the testing process. The obvious reason for this is that the cost of branch divergence is far greater as compared to the cost of atomic operations which are used in the flag-based method. On the contrary, the last method which is proven more efficient theoretically performed badly as compared to the Q-based and flag-based method in terms of speed-up rate. This is due to the reason that our input graph sizes were smaller while the prefix-based approach benefits more as the size of the graph becomes larger and larger. The reason for which we could not input very large graphs is the memory constraints of our GPU as APSP problems consumes a very large amount of memory.

We can observe from figure 9 that as the size of graphs becomes larger and larger, the advantage of using parallel GPU implementation keeps increasing. Until the number of edges about fifteen thousand, no considerable advantage is observed. However as the number of edges increase more than that, the execution time to process each node in the graph starts to decrease.

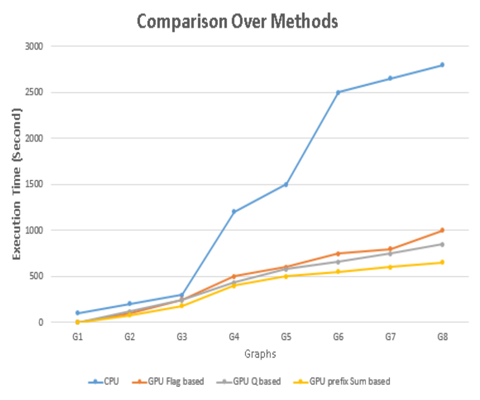


Fig. 8. Execution times on multiple graphs.

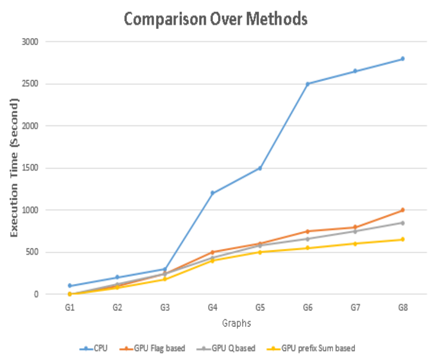


Fig. 9. Speed-up rates on multiple graphs.

9 Conclusions and Discussions

We proposed 3 diverse versions of Johnson’s APSP algorithm on GPU to efficiently find shortest path for all the nodes present in a graph. The results indicates that atomic operations affect the performance of the proposed solution. However, this approach is still faster as compared to flag-based version because of branch divergence problem. Nevertheless, prefix-based approach can perform better if the size of the input is increased substantially.

Another advantage of the proposed solution is that it can work with negative weighted graphs. Moreover, it can perform more efficiently as compared to Floyd-Warshall algorithm [2] if the size of the input is very larger.

In future, it would be interesting to perform GPU implementation of Floyd-Warshall algorithm [2] and compare the resulting algorithm with the proposed solution.

References

[1] Javaid, Adeel. Understanding Dijkstra Algorithm. A Review Paper available on SSRN Electronic Library at https://ssrn.com/abstract=2340905, P 1-27, 2013

[2] Taştan, Oğuzhan & Eryüksel, Oğul & Temizel, Alptekin. Accelerating Johnson's All-Pairs Shortest Paths Algorithm on GPU, A report available at https://github.com/ouzan19/JohnsonAlgoCUDA. P 1-6, 2017.

[3] Xing, Lizhi, and Yujie Li. Revised Floyd-Warshall Algorithm to Find Optimal Path in Similarity-Weight Network and Its Application in the Analysis of Global Value Chain. In Journal of Physics: Conference Series, Vol. 1298, No. 1, P 1-6, IOP Publishing, 2019.

[4] Abu-Ryash, H., and A. Tamimi. Comparison studies for different shortest path algorithms. International Journal of Computers and Applications, Vol 14, No. 8, Pages 5979-5986, 2015.

[5] Walden, David. The Bellman-Ford Algorithm and Distributed Bellman-Ford. P 1-12, A technical report available at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.84.6549, 2005.

[6] Biradar, Anil & Gopala, Harish. A Systolic Solution for Computing the Symmetrizer of a Hessenberg Matrix. Available at https://www.researchgate.net/publication/264849768_A_Systolic_Solution_for_Computing_the_Symmetrizer_of_a_Hessenberg_Matrix, 2021. (Unpublished)

[7] Venkataraman, Gayathri, Sartaj Sahni, and Srabani Mukhopadhyaya. A blocked all-pairs shortest-paths algorithm. Journal of Experimental Algorithmics (JEA), Vol 8, P1-19, 2003.

[8] Klugman SA. Heckman–Meyers Algorithm. Encyclopedia of Actuarial Science. Wiley Online Library, Vol 2, 2006.

[9] Madduri, K., Bader, D. A., Berry, J. W., & Crobak, J. R. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In 2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX), P 23-35. Society for Industrial and Applied Mathematics, 2007.

[10] Berry, Jonathan W., et al. Parallel Shortest Path Algorithms for Solving Large-Scale Instances. No. SAND2006-6307P. Sandia National Lab (SNL-NM), Albuquerque, NM (United States), 2006.

[11] Crauser, A.: LEDA-SM: External Memory Algorithms and Data Structures in Theory and Practice. PhD thesis, Universität des Saarlandes, Saarbrücken, http://www.mpi-sb.mpg.de/~crauser/diss.pdf, 2001.

[12] J. Nickolls and I. Buck. NVIDIA CUDA software and GPU parallel computing architecture. Microprocessor Forum, May 2007.

[13] Han, W.Y. An improvement on fixed order Bellman-Ford algorithm. J. Harbin Inst. Technol. Vol 46, P. 58–62. 2014

[14] Kole, Sebastiaan & Figge, Marc & Raedt, H. Unconditionally Stable Algorithms to Solve the Time-Dependent Maxwell Equations. Physical review. Vol 64, No. 6 (066705), 2002.

[15] Ortega-Arranz H, Llanos DR, Gonzalez-Escribano A. The shortest-path problem: Analysis and comparison of methods. Synthesis Lectures on Theoretical Computer Science. Vol 1, No. 1, P. 1-87. 2014.