

**INTERDISCIPLINARY DOCTORAL SCHOOL**  
**Faculty of Mathematics and Computer**

**Delia Elena CUZA (married SPRIDON)**

# **GPU Methods for Increasing Computational Performance in Graph Theory and Map Building**

**SUMMARY**

**Scientific supervisor**  
**Prof. dr. Marius Sabin TĂBÎRCĂ**

**BRAŞOV, 2024**



## Contents

INTRODUCTION.....	3
Chapter 1. High-Performance Computing on GPU.....	7
1.1. Generalities.....	7
1.2. Parallel Programming on GPU. Applications.....	8
1.3. Technologies for Programming on GPU– CUDA.....	9
Chapter 2. Generation of Random Networks.....	11
2.1. Graphs - General Information.....	11
2.2. CUDA in Graph Theory.....	12
2.3. Algorithms for Generating Random Graphs.....	12
2.4. Results and Discussions.....	17
Chapter 3: Finding the Minimum Loss Path in a Big Network.....	18
3.1. Scientific Context.....	18
3.2. The Minimum Loss Path Problem.....	19
3.3. Algorithms for Determining the Shortest Path in a Network.....	21
3.4. Results and Discussions.....	23
Chapter 4. Determining the Minimum Loss Flow in a Generalized Network.....	25
4.1. The Traditional Maximum Flow Problem.....	25
4.2. The Generalized Maximum Flow Problem.....	26
4.3. Results and Discussions.....	27
Chapter 5. Fast GPU Interpolation for Map Generation.....	28
5.1. Two-Dimensional Interpolation Methods.....	28
5.2. Accelerating Interpolation Methods Using CUDA.....	29
5.3. Study of Air Pollution Maps for Braşov During the Pandemic.....	30
5.4. Study of Geomagnetic Maps of Romania.....	32
5.5. CUDA Methods for Generating Geomagnetic Maps.....	32
Chapter 6. Conclusions and Future Perspectives.....	34
Published Works in the Field of the Thesis.....	36
Selective bibliography.....	37

## INTRODUCTION

In this thesis, I used CUDA (Compute Unified Device Architecture) technology to optimize and enhance specific algorithms in various fields. I have structured the main contributions and results into several major directions, each bringing significant improvements in its application area by using parallel computing capabilities offered by NVIDIA GPUs (Graphics Processing Units). This work is based on six papers published in prestigious scientific journals or presented at international conferences, all indexed in recognized international databases, and one paper accepted for presentation and publication in the proceedings of a conference indexed by CORE.

The first major contribution was the development and implementation of new algorithms for generating random networks, which are essential for modeling and simulating various natural and social phenomena. Random networks are used in numerous applications, from analyzing social structures to simulating diffusion processes in physics and chemistry. Using CUDA technology allowed for a significant acceleration in the process of generating these networks. Compared to traditional CPU (Central Processing Unit)-based methods, the proposed solution reduced execution time by parallelizing the generation operations, resulting in a significant increase in performance and the ability to handle large-scale networks.

The second contribution was the proposal and implementation of algorithms for determining minimum loss paths in generalized networks. These algorithms are critical in various flow optimization applications, such as logistics, transportation, and telecommunications networks. Implementing them on the CUDA platform enabled parallel processing of nodes and edges in the network, significantly reducing computation time. Instead of processing each path sequentially, GPUs allowed simultaneous calculations, leading to much faster optimal solutions. This increased efficiency was demonstrated through tests on complex networks, where CUDA algorithms reduced the time required to determine optimal paths compared to traditional CPU solutions.

As a practical application of the minimum loss path determination algorithms, we proposed a solution for minimizing flow loss in networks. This problem is particularly relevant in the context of distribution networks for energy, water, or other types of networks where losses can occur along the arcs. GPU optimization using CUDA enabled intensive calculations to be performed much faster than classical approaches. The developed algorithms were tested on large-scale networks and demonstrated high efficiency in identifying and minimizing losses. The results showed that the use of GPUs not only accelerates the computation process but also improves overall performance.

Finally, we applied interpolation methods, such as Inverse Distance Weighting (IDW) and kriging, using CUDA to generate precise and detailed pollution and geomagnetism maps in a short time. Interpolation is a crucial method for mapping spatial data, used in geography, meteorology, and other Earth sciences. Implementing these methods on GPUs allowed for the parallelization of distance and weight calculations, significantly speeding up the interpolation process. This acceleration was particularly useful for large and complex datasets where traditional calculations would be too slow.

In conclusion, by using CUDA technology, we optimized and enhanced essential algorithms for various applications, demonstrating that GPUs can bring significant improvements in performance and scalability of these algorithms. This paper highlights the enormous potential of parallel computing on

GPUs in solving complex problems and opens new directions for future research in the field. Thus, the research presented here is not only based on a solid foundation of studies and experiments published and validated internationally but also demonstrates extensive practical applicability in multiple scientific and technological domains.

In summary, this thesis is based on the results obtained and published in journals or proceedings of internationally recognized conferences. Thus, in the domain of the thesis, I have published:

- 1 ISI article in an A-list journal
- 2 articles in a Scopus-indexed journal
- 3 articles presented and published in the proceedings of CORE C-classified conferences
- 1 paper accepted for presentation at a CORE C-classified conference

**Table 1** presents the classification of these works according to the standards for evaluating PhD theses in the field of Computer Science, valid at the time of the thesis defense. Furthermore, in terms of the impact of the results, I highlight that the papers published in the thesis domain have 11 citations (excluding self-citations), of which:

- 4 citations are in ISI-rated journals
- 1 citation in a Scopus-indexed journal
- 2 citations in the proceedings of CORE C-classified conferences
- 3 citations of category D.

**Table 1** *Published Works and Corresponding Scores, According to PhD Thesis Evaluation Standards 2018.10.01-Present<sup>1</sup>, Citations (excluding self-citations)*

No.	No. of authors	Article Title	Journal / Proceeding	International Database	Score <sup>1</sup>	Citations
1.	2	Adaptation of Random Binomial Graphs for Testing Network.	Mathematics	ISI – A	8p	3
2.	1	Advances in CUDA for computational physics	Bulletin of the Transilvania University of Brasov. Series III: Mathematics and Computer Science	Scopus	2p	1

<sup>1</sup> Standarde de evaluare a tezelor de doctorat: <https://www.cs.ubbcluj.ro/invatamant/programe-academice/doctorat/standarde-evaluare-teze-de-doctorat/>

3.	3	IDW map builder and statistics of air pollution in Brasov	Bulletin of the Transilvania University of Brasov. Series III: Mathematics and Computer Science	Scopus	2p	3
4.	3	Fast CUDA Geomagnetic Map Builder	ICCSA - Lecture Notes in Computer Science	CORE C	2p	-
5.	3	Finding minimum loss path in big networks	ISPDC - IEEE Xplore	CORE C	2p	4
6.	4	New approach for the generalized maximum flow problem	IASID – AC	CORE C	1p	-
<b>Total</b>					17p	11

In **Table 2**, we have presented the fulfillment of the current national minimum standards for awarding the title of Doctor in the field of Computer Science. Thus, in the thesis domain, I have published:

- 1 paper in an ISI-rated journal
- 2 papers in a Scopus-indexed journal
- 4 papers presented at international conferences, of which 3 are in ISI, CORE C, Scopus, DBLP, IEEE/Springer, etc.

**Table 2** *Fulfillment of National Minimum Standards for Awarding the Title of Doctor – Computer Science Committee<sup>2</sup>*

No.	Criteria	Type of paper	No. of papers published by the author
1.	Publication or acceptance for publication (with proof of acceptance) of at least one article in ISI-indexed journals from the UEFISCDI list or in SCOPUS-indexed journals.	ISI	1
		Scopus	2
2.	Participation in and presentation of at least two scientific papers at international		4



	conferences, as proven by the conference program.		(of which 3 ISI, CORE C, Scopus, DBLP, IEEE/Springer etc.)
3.	Recognized conferences are those indexed in the following databases: SCOPUS, IEEE, ACM, SPRINGER, DBLP, CiteSeerX, Zentralblatt, MathSciNet, COPERNICUS, EBSCO, and ProQuest.		

# Chapter 1. High-Performance Computing on GPU

In this chapter, a literature review published in my work (Spridon, Advances in CUDA for Computational Physics 2023) is presented. It provides a summary of the most important research results from recent years regarding GPU programming. Additionally, the most well-known methods of GPU programming are compared, highlighting the advantages and limitations of GPU programming using CUDA technology.

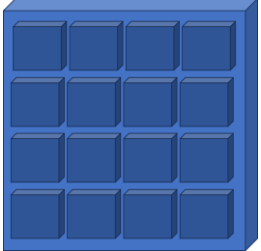
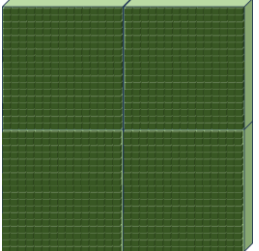
## 1.1. Overviews

High-Performance Computing (HPC) is a field of computer science that focuses on using systems and technologies to perform complex or computationally intensive calculations at superior speeds and efficiency. This field often deals with solving difficult problems and handling massive amounts of data in the shortest possible time. One of the most accessible methods to achieve this is by using Graphics Processing Units (GPUs).

Table 1.1 provides a comparative overview of CPUs and GPUs. In short, GPU programming offers many benefits, including parallel processing, energy efficiency, cost-effectiveness, and flexibility. However, it also requires specialized knowledge and experience, has additional overhead for data transfer, and does not apply to all types of applications. Additionally, the performance gains of GPU programming are limited by hardware constraints, and large-scale applications may require specialized hardware or multiple GPUs to achieve optimal performance.

The efficiency of a GPU can be directly proportional to the number of GPU cores. Due to this, GPUs can fully benefit from Moore's Law or the constant increase in integration density. GPU performance continues to improve at a rate of 1.5 times per year. In 2017, the performance gain over CPUs was 10-100 times, depending on the application. By 2025, this is estimated to be nearly 1,000 times. Thus, while Moore's Law has slowed down for CPUs, and some even say it has ended, the growth in GPU computing power continues to keep pace (Huang 2023).

Table 1.1. CPU / GPU comparison

CPU	GPU
	
Up to several dozen very powerful cores	Up to several thousand cores optimized for parallelism
Higher frequencies for fast instruction execution	Relatively lower frequencies but efficient parallel operations

Larger and more efficient cache memory for general processing tasks	Smaller cache memory optimized for large datasets specific to graphics, in general
Ideal for single-thread or lightly multi-threaded compute tasks	Optimized for graphics, parallel processing, and massively parallel algorithms
Lower power consumption, ideal for portable systems	Higher power consumption
Usually more expensive per core, but the price can vary based on performance	More affordable per core, but the total cost can be higher depending on configuration and graphics performance
Executes instructions for general processing	Executes parallel operations for graphics and intensive computation

It is important to emphasize that CPUs and GPUs are designed for different uses, and the choice between them depends on the type of tasks that need to be performed.

### 1.2. *Parallel Programming on GPU. Applications.*

Due to their high parallel processing power, GPU programming has found applications in a wide range of fields. Some of the areas where GPU programming is used in recent research include artificial intelligence (AI) and machine learning (ML), big data analysis, scientific simulations, graphics and 3D rendering, medicine and bioinformatics, or cryptography and security (Figure 1.1).



Figure 1.1 *Recent Applications of GPU Programming (Baji 2018)*

GPUs enable the rapid transformation and analysis of large datasets (big data) (Chen et al. 2018). This process includes real-time data analysis, data processing and filtering, and the application of machine learning algorithms to large datasets. Consequently, there is research exploring how GPUs can be used to accelerate big data processing. For example, various parallelization and optimization techniques are analyzed to achieve high performance in big data analysis (Wu, Sun et al. 2021) (Kumar and Mohbey 2022). Algorithms and optimization techniques are also proposed to reduce execution time and efficiently manage memory in big data analysis operations (Jiang et al. 2015).



GPU programming is used in computational sciences to accelerate intensive numerical computations (Prabhu et al. 2011). This includes simulations in physics, chemistry, biology, and other fields where complex and iterative calculations are performed. In computational physics, for instance, process acceleration is of great importance for obtaining the desired results in real time. GPU programming is a suitable approach for achieving excellent execution times when massive parallelization is possible (Spridon, Advances in CUDA for Computational Physics 2023). Thus, although many known algorithms used in computational physics have already been parallelized and some of them are included in the CUDA library (NVIDIA 2019), new methods of optimization and speed enhancement are still being sought. Execution time is crucial in many computational physics problems, and therefore any improvement in this direction is still necessary. Hybrid parallel algorithms (CPU-GPU) are continuously developed to achieve high-performance computing results with minimal costs (Spridon, Advances in CUDA for Computational Physics 2023).

### 1.3. Technologies for Programming on GPU– CUDA

There are several technologies and platforms available for GPU programming. Among these, the most important are CUDA (Compute Unified Device Architecture), OpenCL (Open Computing Language), SYCL (Single-source Heterogeneous Programming in C++), and Vulkan.

In the literature, there are several studies comparing GPU programming technologies. For example, Karimi et al. perform performance tests and compare data transfer times to and from the GPU, kernel execution times, and end-to-end application execution times for both CUDA and OpenCL on the same graphics card (Karimi, Dickson, and Hamze 2010). Their results are shown in Figure 1.2. As observed from these tests, CUDA performed better in data transfer to and from the GPU. No significant change was noted in the relative performance of data transfer for OpenCL when transferring larger amounts of data. CUDA kernel execution was also faster than OpenCL, even though the two implementations were very similar.

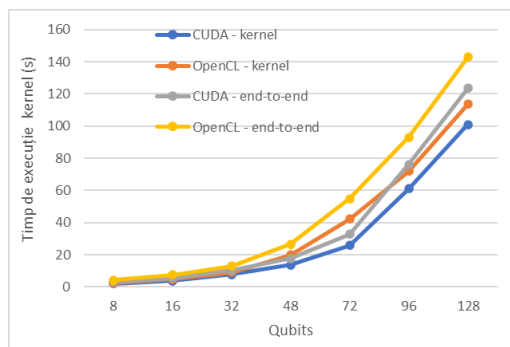
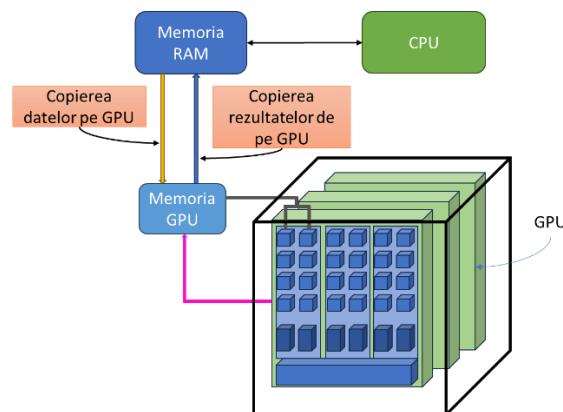


Figure 1.2 Comparison of CUDA vs. OpenCL Execution Times

Thus, it is demonstrated that the CUDA architecture is a better choice for applications requiring high performance. Otherwise, the choice between CUDA and other GPU programming technologies can be made by considering factors such as previous familiarity with any of the systems, available development tools for the target GPU hardware, or the portability of the resulting application. In this work, I chose CUDA architecture for its superior performance previously demonstrated in the literature.

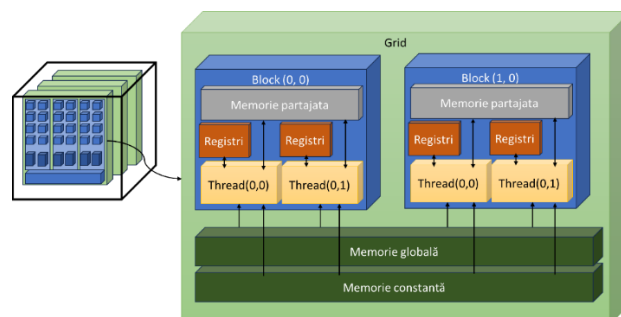
A schematic of a CUDA workflow protocol is presented in **Figure 1.3**. Applications begin running on the CPU, and the host code manages both host and device code. Data to be processed is loaded into host memory, necessary memory is allocated on the device, and data is loaded into device memory using CUDA API calls such as `cudaMalloc()` or `cudaMemcpy()`.

Kernel functions are called from the CPU and executed on the GPU, leveraging the GPU's capability to handle intensive tasks that can be executed in parallel. To launch a kernel function, the number of threads and blocks to be used must be specified. This is done using the `<<<>>>` syntax in CUDA. Once the kernel is launched, it executes on the GPU. Each thread will execute the same code but with different data. Data for each thread is accessed using the thread index, which is provided by CUDA. To ensure that all threads have completed their work before moving to the next step, threads need to be synchronized using the `__syncthreads()` function. After the kernel has finished executing, data must be transferred back from the GPU to the CPU. Finally, memory allocated on the device must be freed using `cudaFree()`.



**Figure 1.3** *CUDA Program Workflow Steps*

Memory management plays a crucial role in achieving the best results with CUDA programming. It is also necessary to understand the GPU memory hierarchy so that it can be utilized as efficiently as possible. The GPU memory levels (global memory, constant memory, shared memory, local memory, and registers) are illustrated in Figure 1.4.



**Figure 1.4** *GPU Memory Hierarchy*

In the field of parallel computing and applications requiring high computing power, CUDA has become a popular technology. With CUDA, programmers can leverage the massive processing power

of GPUs to accelerate the solving of complex problems and achieve superior performance across various domains.

## Chapter 2. Generation of Random Networks

This chapter presents two methods for generating random networks, which are necessary for studying the efficiency of algorithms in graph theory. The proposed generation methods are parallelized, and the results regarding execution times and acceleration using CUDA programming are discussed. This chapter builds on the work by Deaconu and **Spridon** (2021), to which I am a co-author.

### 2.1. *Graphs - Fundamentals*

Graphs constitute a significant branch of both mathematics and computer science, concentrating on the analysis of structures that illustrate relationships between objects. They are used to model and analyze interconnections between various entities or elements. A graph consists of a collection of nodes or vertices, represented by points, and edges or arcs that connect these nodes. Graph theory studies the properties, characteristics, and algorithms associated with graphs. Graphs are widely used in different fields, including computer science, networks, optimization, artificial intelligence, and bioinformatics. Graph algorithms are used to solve problems related to search, traversal, connectivity, scheduling, and many others. By studying and applying graph theory, we can understand and analyze complex structures of relationships between objects, find efficient solutions to various problems, and develop optimized algorithms for different scenarios.

Graph theory is a field that investigates the characteristics and behavior of various types of graphs, as well as the development of specialized algorithms to solve problems associated with graphs, being a branch of discrete mathematics.

**Definition 2.1** A graph is an ordered pair,  $G = (V, E)$ , consisting of a set  $V$  of elements called nodes or vertices and a set  $E$  of edges (or arcs) that connect these nodes. The formal definition of a graph can vary depending on the context in which it is used, but the following presents some basic elements of graph theory.

In a graph  $G = (V, E)$  the number of elements in  $V$  or the cardinality of the set  $V$  is called the order of  $G$ , and the number of elements in  $E$ , or the cardinality of the set  $E$ , is called the size of  $G$ . The order of a graph is usually denoted by  $n$ , and the size of  $G$  is denoted by  $m$ . Each element in  $V$  is called node (or vertex), and each element in  $E$  is called edge. For an arc  $a = (u, v)$ , the node  $u$  and node  $v$  are adjacent nodes; the arc  $a$  and node  $u$  (or  $v$ ) are incident to each other. For each arc  $a = (u, v)$ , the nodes  $u$  and  $v$  are called terminal nodes. A loop is an arc  $a = (u, v)$  whose terminal nodes are identical, i.e.,  $u = v$ . Multiple edges are a set of edges that have the same pair of terminal nodes.

**Definition 2.2** A random graph is a graph where the number of nodes, the number of edges, and the connections between them are generated randomly through various methods.

sErdős and Rényi introduced binomial random graphs in their 1959 paper (Erdős and Rényi 1959). These random graphs are generated based on the values of two parameters:  $n$  (the number of nodes) and  $p \in [0,1]$  - the probability of introducing any edge into the graph. In a network generated in this way, there is a possibility that the source might poorly communicate with the storage node or even not communicate at all. An algorithm for generating simple random graphs with a given degree sequence was developed in a paper by Bayati et al. (Bayati, Kim, and Saberi 2010). Using this algorithm, a random uniform graph with a given degree sequence is generated very quickly (in almost linear time). In 2002, Albert and Barabási introduced their model (BA), consisting of an algorithm based on the preferential attachment mechanism for generating scale-free random networks (Albert and Barabási 2002). Networks generated in this way have real-world applications on the Internet, citation networks, the World Wide Web, and some social networks. The algorithm starts with a network having  $m_0$  given nodes. Sequentially, nodes are introduced into the network. Each of these newly added nodes is connected to  $m \leq m_0$  existing nodes using a given probability, which is proportional to the number of connections that the previously added nodes already had. The probability  $p_i$  of connecting a new node to node  $i$  is:

$$p_i = \frac{k_i}{\sum_j k_j} \quad (2.1)$$

Given that existing results in the literature about networks deal with specific graphs that are not general enough or inadequate for network flow problems, in the work (Deaconu and Spridon 2021), we proposed a new idea for generating random networks that has the advantages of being fast and based on the natural property of flow, which can be decomposed into elementary directed paths and cycles. Consequently, networks generated in this way are suitable for testing the correctness and efficiency of algorithms for network flow problems, such as minimum cost flow, maximum flow, multi-commodity flow problem, etc.

## 2.2. CUDA in Graph Theory

GPU programming has shown promising results in accelerating graph theory algorithms in recent years. Some of the most recent research in GPU programming for graph theory is in the following directions: Graph Neural Networks (GNN) (Zonghan Wu, 2021) (Tianfeng Liu, 2023), graph partitioning (Santosh Nage, 2015), triangle counting in a graph (Liu Hu, 2021), algorithms for finding the shortest path between two nodes in a graph (Carl Yang, 2022).

In general, recent research in GPU programming for graph theory demonstrates the potential of GPUs to accelerate graph theory algorithms and handle very large graphs. This can lead to improved performance and scalability for a wide range of applications, from machine learning to social network analysis or routing problems.

## 2.3. Algorithms for Generating Random Graphs

Let  $G = (V, E, s, t, c, w)$  be an  $s - t$  network, where  $V$  is a set containing  $n > 0$  vertices (nodes), and  $E$  is a set of  $m \geq 0$  so-called arches (directed edges). Each arch  $a = (u, v) \in E$  connects two nodes  $u$  and  $v$  from  $V$ , and  $s$  is a special node called the source and  $t$  is a node called the sink. In  $G$ , we define the capacity function  $c: E \rightarrow R_+^*$  and the cost function  $w: E \rightarrow R_+$ . The value  $c(a)$  is the

maximum flow that can be transported from node  $u$  to node  $v$  on edge  $a = (u, v) \in E$ , and  $w(a)$  is the unit cost of transporting flow on edge  $a$ .

Ahuja and co-authors present the following theorem (Ahuja, Magnanti, and Orlin 1993):

**Theorem 2.1** Any admissible flow can be decomposed into paths and circuits such that:

- (a) Any path with positive flow connects the source  $s$  to the sink node  $t$ .
- (b) At most  $n + m$  paths and cycles have non-zero flow. Of these, at most  $m$  cycles have non-zero flow.

The proof of Theorem 2.1 can be found in (Ahuja, Magnanti, and Orlin 1993).

Comparisons of the correctness and efficiency of algorithms for flow problems are important when developing new methods to solve them. To achieve this, a fast and reliable tool is needed to generate random networks, starting from simple ones and extending to large-scale networks. We developed a method based on the Erdős–Rényi model using the idea from **Theorem 2.1** to create such a tool. Since a flow can be decomposed into elementary flows, a natural approach is to generate random  $s - t$  paths and elementary cycles. In the work by Deaconu and Spridon (2021), algorithms are presented for generating  $s - t$  paths and elementary cycles in a network. Thus, a primary algorithm to generate a random  $s - t$  path in a network with  $n$  nodes is **Algorithm 2.1**.

Deaconu and Spridon propose algorithms for generating paths and elementary circuits in a network (Deaconu and Spridon 2021). Thus, a primary algorithm for generating a random  $s - t$  elementary path in a network with  $n$  nodes is **Algorithm 2.1**.

---



---

### Algorithm 2.1. Algorithm Random s-t Directed Elementary Path v1 (ARDEP1)

---



---

```

/* source is considered having the first index, and sink is considered having the last one */
s = 0;
t = n - 1;
/* only source is initially part of the path */
for each node j other than s do
    pathnode[j] = false;
end for;
pathnode[s] = true;
/* build the random path */
u = s;
for j = 1 to n - 1 do
    /* choose a random index k of the next node to be added to the path */
    k = random(0, n - j - 1);
    l = 0;
    /* find node v as the k-th node out of the nodes not before chosen */
    for each node v do
        if pathnode[v] then
            continue;
        end if;
        if l = k then
            break;
        end if;
        l = l + 1;
    end for;
    /* add arc (u, v) to the network */
    ma[u][v] = 1;

```

---

---

```

/* mark node v as being part of path */
pathnode[v] = true;
/* if the last node v added to the path is sink, then path is completed */
if v = t then
    break;
end if;
/* node u becomes v to prepare the adding of another node to the path */
u = v;
end for;

```

---

In ARDEP1, without restricting the generality of the algorithm, we consider the source's index equal to 0, and  $n - 1$  as the index of the sink node  $t$ . The algorithm builds a path starting from  $s$ . At each iteration, a new node that was not previously added to the path is randomly selected and pushed at the end of the path. Each time a new node  $v$  is pushed back to the path, the arc  $(u, v)$  is added to the network, i.e., the value of the adjacency matrix  $ma$  is set to 1 on the position  $(u, v)$ , where  $u$  is the node previously added to the path. The algorithm ends when the sink node is added to the path.

For generating a random circuit (**Algorithm 2.2**), the algorithm is presented below.

---

### Algorithm 2.2. Algorithm Random Directed Elementary Cycle v1 (ARDEC1)

---

```

/* choose a random node u0 */
u0 = random(0, n - 1);
/* only node u0 is initially part of the cycle */
for each node j other than u0 do
    cyclenode[j] = false;
end for;
/* build the random cycle */
u = u0;
for j = 0 to n - 1 do
    /* choose a random index k of the next node to be added to the cycle */
    k = random(0, n - j - 1);
    l = 0;
    /* find node v as the k-th node out of the nodes not before chosen */
    for each node v do
        if cyclenode[v] then
            continue;
        end if;
        if l = k then
            break;
        end if;
        l = l + 1;
    end for;
    /* if v is u then regenerate v. This can only happen when u = u0 */
    if u = v then
        j = j - 1;
    else
        /* add arc (u, v) to the network */
        ma[u][v] = 1;
        /* mark node v as being part of cycle */
        cyclenode[v] = true;
    end if;
    /* if v is the first chosen node u0, then cycle is completed */
    if v = u0 then
        break;

```

---

---

```

end if;
/* node u becomes v to prepare the adding of another node to the cycle */
u = v;
end for;

```

---

In ARDEC1, a cycle is built starting with a randomly chosen node  $u_0$ . At each iteration, a new node that is not already part of the cycle is randomly selected and added to the cycle. Each time a new node  $v$  is introduced into the cycle, the arc  $(u, v)$  is also added to the network, where  $u$  is the node previously added to the cycle. The algorithm ends when the node  $u_0$  is added again to the cycle.

The algorithms ARDEP1 and ARDEC1 can naturally build directed elementary  $s - t$  paths and cycles. Their time complexity is obviously  $O(n^2)$ . These two algorithms could be used together to build random networks. However, we shall present a faster approach below.

Richard Durstenfeld proposes an algorithm to randomly generate a permutation (Durstenfeld 1964). In **Algorithm 2.3**, we propose a similar but simpler approach to generate a shuffled vector of nodes having the indexes between  $istart$  and  $iend$  (Deaconu and Spridon 2021).

---

### Algorithm 2.3. Algorithm Shuffled Vector of Nodes (ASVN)

---

```

Input:  $istart, iend$ ;

/* the vector "nodes" initially contains the indexes from  $istart$  to  $iend$  */
for  $j = istart$  to  $iend$  do
    nodes[j] = j;
end for;
/* shuffle the vector "nodes" */
for  $k = istart$  to  $iend$  do
     $u = random(istart, iend)$ ;
     $v = random(istart, iend)$ ;
    if  $u = v$  then
        swap = nodes[u];
        nodes[u] = nodes[v];
        nodes[v] = swap;
    end if;
end for;

```

---

Next, I present two novel methods for randomly generating directed elementary  $s - t$  paths and cycles using ASVN.

---

### Algorithm 2.4. Algorithm Random $s - t$ Directed Elementary Path v2 (ARDEP2)

---

```

/* efficiently generate a shuffled vector of nodes without  $s$  and  $t$  */
ASVN(1,  $n - 2$ );
 $s = 0$ ;
 $t = n - 1$ ;
/* randomly generate the length of the path */
 $lpath = random(2, n)$ ;
/* add the arcs given by the first  $lpath$  nodes of the shuffled vector to the network */
 $ma[s][nodes[1]] = 1$ ;
for  $k = 1$  to  $lpath - 3$  do
     $ma[nodes[k]][nodes[k + 1]] = 1$ ;
end for;

```

---

---



---

```
ma[nodes[lpath - 2]][t] = 1;
```

---

In **Algorithm 2.4**, first, ARDEP2 randomly generates the length of the path.  $lpath - 2$  nodes are then taken from the shuffled vector of nodes, and together with source and sink, generate the path.

---



---

### Algorithm 2.5 Algorithm Random Directed Elementary Cycle v2 (ARDEC2)

---



---

```
/* efficiently generate a shuffled vector of nodes */
ASVN(0, n - 1);
/* randomly generate the length of the cycle */
lcycle = random(2, n);
/* add the arcs given by the first lcycle nodes of the shuffled vector to the network */
for k = 0 to lcycle - 2 do
    ma[nodes[k]][nodes[k + 1]] = 1;
end for;
ma[nodes[lcycle - 1]][nodes[0]] = 1;
```

---

In **Algorithm 2.5**, ARDEC2 takes  $lcycle$  nodes from the shuffled vector of nodes and generates a cycle.

Below, I introduce **Algorithm 2.6** for generating a random flow network.

---



---

### Algorithm 2.6. Algorithm Generating Random $s - t$ Flow Network (AGRFN)

---



---

```
/* generate npath random paths */
for k = 1 to npath do
    ARDEP2;
end for;
/* generate ncycle random cycles */
for k = 1 to ncycle do
    ARDEC2;
end for;
/* generate the adjacency lists la using the adjacency matrix ma */
for i = 0 to n do
    la[i] = null;
end for;
/* randomly attach capacities and costs to the arcs when they are added to la */
for i = 0 to n do
    for j = 0 to n do
        /* generate arcs according to Erdos-Rényi model */
        if ma[i][j] = 0 and random(0, 1000) < p * 1000 then
            ma[i][j] = 1;
        end if;
        if ma[i][j] = 1 then
            Push back (j, random(minu, maxu), random(minc, maxc)) to la[i];
        end if;
    end for;
end for;
```

---

**Theorem 2.3.** The time complexity of AGRFN is  $O(n \cdot \max\{n_{path}, n_{cycle}, n\} / g)$ .

Usually, it is enough to consider the number of paths and the number of cycles less than the number of nodes. So, in practice, the time complexity is likely to be  $O(n^2)$ .

The time complexity from **Theorem 2.3** can be improved if the generation of the paths, cycles, and adjacency lists are parallelized. The computations from the algorithm are elementary and they only

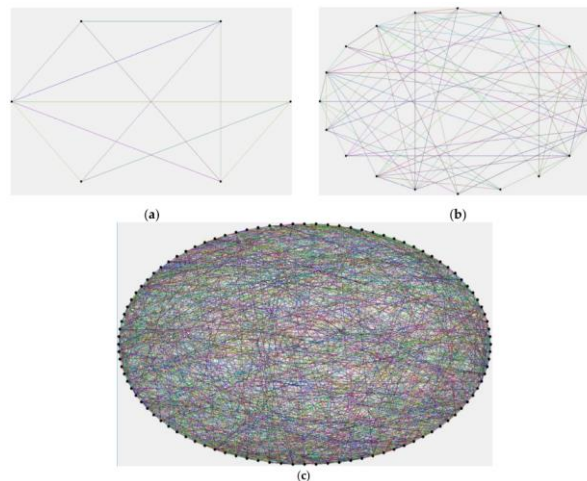


involve integer values. So, AGRFN can be naturally parallelized on GPUs. Since the speed of generating of large-scale random networks is essential, time complexity improvement by parallelization can act an important role. Considering a total of  $g$  GPU cores, the generation of the paths and cycles can be divided into  $\max\{1, (n_{path} + n_{cycle}) / g\}$  groups. The generation of the adjacency lists can also be divided into  $\max\{1, n/g\}$  groups. So, the time complexity of the parallel implementation on GPUs of AGRFN is  $O(n \cdot \max\{n_{path}, n_{cycle}, n\} / g)$ .

## 2.4. Results and Discussions

In **Figure 2.1**, three networks having 6, 20, and 100 nodes, respectively, were generated and displayed. For the first network, 3 paths and 2 cycles were generated. For the second network, 10 paths and 2 cycles were generated, and for the last network, 20 paths and 10 cycles were generated.

Different tests were performed to illustrate the generating time of increasing the scale of random networks having the number of nodes between 10 and 10,000. As expected, and as shown in Table 1, the number of nodes together with the number of considered paths and cycles directly influence the speed of the network generation. An Asus ROG Strix G17 G712LV, Intel Core i7 - 10750H up to 5.10 GHz processor, 16GB RAM, NVIDIA GeForce RTX 2060 6GB GDDR6 with 1920 CUDA cores was used.



**Figure 2.1.** Networks generated using AGRFN. (a) Network with  $n = 6$ ,  $n_{path} = 3$ ,  $n_{cycle} = 2$ ; (b) network with  $n = 20$ ,  $n_{path} = 10$ ,  $n_{cycle} = 2$ ; (c) network with  $n = 100$ ,  $n_{path} = 20$ ,  $n_{cycle} = 10$ .

The parallelization was implemented using CUDA programming on GPU. Each path and cycle were created on a different thread. Additionally, the creation of adjacency lists from the adjacency matrix was parallelized, the list for each node being obtained on a different thread. For small networks (less than 50 nodes) it is better to use the implementation of the algorithm on CPU, but when the number of the nodes of the networks is more than 50, the CUDA implementation is preferred resulting in a clear speed-up, up to 19 times faster than the CPU implementation. The speed-up was calculated as the ratio between CUDA and CPU execution times. The best speed-up was obtained for large-scale networks having thousands of nodes.

In **Figure 2.2**, the speed-up evolution for generating networks of different dimensions is presented. As can be observed, for small-sized networks, running on the GPU leads to a decrease in

execution speed, most likely due to communication times between the CPU and GPU. As the network size increases, the acceleration factor due to massive parallelization on the GPU also increases, reaching an execution time 19 times shorter for a network with 10,000 nodes when running using CUDA.

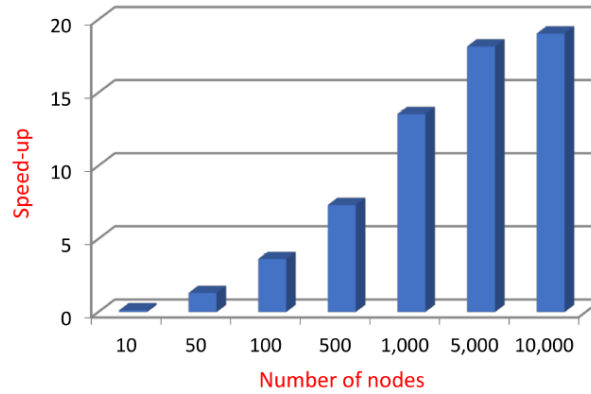


Figure 0.2 CPU/CUDA speed-up (Deaconu and Spridon 2021)

The analysis of speed-up evolution for generating random networks of different sizes when using CUDA shows how the system's performance varies depending on the problem size. As expected, the larger the problem size, the more evident the advantages of parallelization with CUDA become. This is due to the GPU's ability to process a large amount of data in parallel, which can lead to significant improvements in execution time compared to sequential implementations on the CPU.

## Chapter 3: Finding the Minimum Loss Path in a Big Network

In this chapter, I introduce and solve a practical problem known as the minimum loss path problem or the maximum delivery rate path. This problem involves finding the path from a source node,  $s$ , to another given node,  $t$ , called the sink, in a generalized network, which has a gain/loss factor attached to each arc, so that the loss is minimal among all  $s - t$  paths. This is based on the work by (Deaconu, Spridon and Ciupala 2023), to which I am a co-author.

### 3.1. Scientific Context

The classic maximum flow problem involves finding the maximum flow that can be transported from a source node to a so-called sink node in a network where each arc has a capacity  $c(i, j)$ ,  $(i, j) \in E$ . For example, a natural gas supply company may want to maximize the amount of natural gas sent between two cities through its pipeline network. Each pipeline in the network obviously has a limited capacity.

In the generalized problem, each arc, in addition to the corresponding capacity, may also have a loss or gain factor that must be considered when calculating the maximum flow. In other words, the generalized problem for determining the maximum flow is an extension of the classic maximum flow problem in a network where, to determine the maximum amount of flow, other factors such as costs or variable capacities of the arcs must also be taken into account.

The Inverse Generalized Maximum Flow (IGMF) problem was introduced and studied by (Tayyebi and Deaconu 2019). In this problem, the goal is to modify the capacities of the arcs so that a certain admissible flow becomes the maximum flow in the modified network, and the distance between the initial capacity vector and the modified capacity vector is minimized.

To the best of my knowledge, these two problems are the only optimization problems that have been analyzed in networks with gains or losses on the arcs.

### 3.2. The Minimum Loss Path Problem

Let  $G = (V, E)$  be a directed graph, where  $V$  is a finite set of elements called vertices or nodes, and  $E$  is a set of ordered pairs of vertices, called arcs or directed edges ( $E \subseteq V \times V$ ). We will consider that graph  $G$  is a mathematical representation of a real-life transportation network (for water, sewage, gas, electricity, etc.), where the arcs represent the transportation lines, and the nodes represent the intersections of these lines. In real-life transportation networks, there are usually losses on the arcs due to various reasons such as evaporation, leaks, energy dissipation, theft, etc. To mathematically model this, we consider for each arc  $(u, v) \in E$  a loss coefficient, or delivery rate, denoted by  $\alpha(u, v) \in (0, 1]$ . Thus, if  $x$  units enter from node  $u$  on arc  $(u, v)$  then  $x \cdot \alpha(u, v) < x$  units will reach node  $v$ .

In the following, I will present a method for calculating the minimum loss path (MLP) or maximum delivery rate path (MDP) from a source node,  $s$ , to a sink node,  $t$ . We denote this problem as the Minimum Loss Path Problem (MLPP). To solve the MLPP, we need to identify a path in  $G$  from  $s$  to  $t$  such that the delivery rate from  $s$  to  $t$  is the maximum among all paths in  $\Pi$ , that is, we need to find the solution to the following optimization problem:

$$\begin{cases} \max\{\alpha(v_0, v_1) \cdot \alpha(v_1, v_2) \cdot \dots \cdot \alpha(v_{k-1}, v_k)\} \\ P = (v_0 = s, v_1, \dots, v_k) \in \Pi \end{cases} \quad (3.1)$$

To solve the above problem, we proceed by transforming problem (3.2) into a minimization problem as follows:

$$\begin{cases} \min\{-\log(\alpha(v_0, v_1) \cdot \alpha(v_1, v_2) \cdot \dots \cdot \alpha(v_{k-1}, v_k))\} \\ P = (v_0 = s, v_1, \dots, v_k) \in \Pi \end{cases} \quad (3.2)$$

where the base of the logarithm is greater than 1, for example, the base can be  $e$  or 10.

The previous problem can be rewritten in the form:

$$\begin{cases} \min\{\beta(v_0, v_1) + \beta(v_1, v_2) + \dots + \beta(v_{k-1}, v_k)\} \\ P = (v_0 = s, v_1, \dots, v_k) \in \Pi \end{cases} \quad (3.3)$$

where:

$$\beta(v_i, v_{i+1}) = -\log(\alpha(v_i, v_{i+1})) \geq 0, i = 0, 1, \dots, k - 1.$$

Given that the  $\beta$  values attached to the arcs are positive, it is now easy to observe that problem (3.2) has been reduced to a classic optimization problem for finding the shortest path in the network  $G = (V, E, \beta)$ . This problem can be efficiently solved using Dijkstra's algorithm with a time complexity of  $O(n^2)$  or  $O(m \cdot \log(n))$ , depending on the implementation (Schrijver 2012) (Fredman and Tarjan 1984), where  $n$  denotes the number of nodes ( $n = |V|$ ), and  $m$  represents the number of arcs ( $m = |E|$ ). Consequently, **Algorithm 3.1** presented below calculates the MLP in GGG.

---

#### Algorithm 3.1. MLP computation in lossy networks

---

---

Input:

- a directed graph  $G = (V, E)$
- $\alpha(u, v) \in (0, 1], (u, v) \in E$

Output:

- MLP of  $G$

/\* Calculate  $\beta(u, v)$  \*/

**For**  $k = 0, m - 1$  **do**

$$\beta(a_k) = -\log(\alpha(a_k)), a_k \in E$$

**end for**

**If** node  $t$  is not reachable from  $s$  **then**

MLPP has no solution

**else**

Apply shortest Path Algorithm starting from  $s$  in  $G = (V, E, \beta)$

Let  $P = (s = v_0, v_1, \dots, v_k = t)$  be the shortest path from  $s$  to  $t$ , then  $P$  is MLP of  $G = (V, E)$  from  $s$  to  $t$

**End if.**

---

We will now investigate the more general case where some arcs may have gains instead of losses. These gains can be obtained, for example, through injection into the network on certain arcs (a new gas source, a prosumer in the electrical network, etc.). Thus, instead of setting  $\alpha(u, v) \in (0, 1]$ , we could consider  $\alpha(u, v)$  as having positive values, with  $\alpha(u, v) > 1$  on an arc  $(u, v)$  if and only if there is a gain on that arc.

This optimization problem has the same mathematical model (3.1) and can also be transformed into the minimization problem (3.3). However, it is observed that the values of  $\beta(a_k) = -\log(\alpha(a_k))$  can be negative (on arcs where  $\alpha(u, v) > 1$ ). Moreover, if there is a negative cycle in the resulting network, it corresponds to a circuit with infinite gain (the product of the  $\alpha$  values on such a cycle is greater than 1). On an  $s - t$  path containing such a circuit, a maximum delivery rate cannot be found because the flow can be infinitely increased by passing infinitely many times through that circuit.

**Algorithm 3.2** solves the MLPP in the generalized case (in networks where there can be both gains and losses on arcs).

---

### Algorithm 3.2 The Minimum Loss Path Determination Algorithm in a Generalized Network

---

Input:

- a directed graph  $G = (V, E)$
- $\alpha(u, v) \in (0, 1], (u, v) \in E$

Output:

- MLP of  $G$

/\* Calculate  $\beta(u, v)$  \*/

**For**  $k = 0, m - 1$  **do**

$$\beta(a_k) = -\log(\alpha(a_k)), a_k \in E$$

**end for**

**If** node  $t$  is not reachable from  $s$  **then**

MLPP has no solution

**else**

Apply Bellman-Ford's algorithm starting from  $s$  in  $G = (V, E, \beta)$

**If**  $G$  has negative cycle **then**

MLPP has not solution.

**else**

Let  $P = (s = v_0, v_1, \dots, v_k = t)$  be the shortest path from  $s$  to  $t$ , then  $P$  is MLP of  $G = (V, E)$  from  $s$  to  $t$

**End if**

**End if**

---

As with the previous algorithm, in Algorithm 3.2, the  $\beta$  function is initially calculated for each arc of the network. Subsequently, the feasibility test for MLPP is performed. If the sink node  $t$  is accessible from the source node  $s$ , the Bellman-Ford algorithm is applied to determine the shortest path in the newly formed network  $G = (V, E, \beta)$ . If a negative cycle is identified in the network, then the problem of finding the minimum loss path has no solution; otherwise, the found path is also the minimum loss path in the initial network.

### ***3.3. Algorithms for Determining the Shortest Path in a Network***

Algorithms for determining the minimum path are methods used in graph theory and operational computing to find the shortest path between two points (nodes) in a graph. These algorithms are essential in various fields such as communication networks, transportation, logistics, and artificial intelligence.

*Dijkstra's algorithm* is designed to find the shortest path in a graph with only positive arc costs (Dijkstra, 1959). It is widely used in computer science and engineering to find optimal paths in transportation networks, internet routing, and many other applications. However, Dijkstra's algorithm cannot be applied to networks with negative arc values.

The *Bellman-Ford algorithm* can determine the shortest path in a graph containing arcs with negative cost values (Bellman, 1958). Additionally, this algorithm can also decide whether the network contains negative cycles (with infinite gains). If such cycles are present, the problem is infeasible.

Considering the size of networks in real-life applications, the execution time of the proposed algorithms is very important. Therefore, a method of parallelization using GPU programming through CUDA (Compute Unified Device Architecture) was used for the proposed algorithms. The Dijkstra and Bellman-Ford algorithms have previously been implemented on CUDA architecture (Harish and Narayanan, 2007; Ortega-Arranz et al., 2013; Surve and Shah, 2017). Various parallelization techniques have been used, resulting in significant speed improvements compared to CPU implementations. We have adapted these approaches for calculating MLPs.

In Dijkstra's algorithm, in each iteration  $i$ , the minimum distance between the source and the nodes belonging to the set of unset nodes (nodes for which the minimum distance has not yet been determined),  $U_i$ , is calculated. One of the nodes for which the distance is equal to this minimum value is set and becomes the node to be analyzed. The outgoing arcs of the analyzed node are traversed to relax the distances corresponding to adjacent nodes.

To parallelize Dijkstra's algorithm, it is necessary to identify which nodes can be used as analyzed nodes simultaneously. There are several studies in which the set of analyzed nodes has been determined in various ways. For example, in the study by Martin, Torres, and Gavilanes (2009), it is proposed to insert all nodes that have a distance equal to the minimum distance into the set of analyzed nodes. Ortega-Arranz et al. propose an improvement by adding nodes that have a distance greater than the determined minimum distance to the set of analyzed nodes (Ortega-Arranz et al., 2013).

The algorithm calculates, in each iteration  $i$ , for each node in the set of unset nodes  $u \in U_i$ , the sum of the distance calculated up to that point and the costs of its outgoing arcs. Subsequently, the minimum of these calculated values is determined. Finally, those nodes whose distance is less than

or equal to this minimum value determined in the previous step are set and inserted into the set of analyzed nodes.  $\Delta_i$  is defined as the minimum value calculated in each iteration  $i$ , which supports that any unset node  $u$  with a distance  $\delta(u) \leq \Delta_i$  can be set. The larger the value of  $\Delta_i$ , the more parallelism is exploited. However, depending on the graph being processed, using a very optimistic  $\Delta_i$  can lead to computations that negate any performance gains over sequential execution.

**Algorithm 3.3** represents the pseudocode of the parallelized Bellman-Ford algorithm for GPU implementation. The first stage is the initialization, which takes place on the GPU. This is followed by the relaxation stage, in which it is checked whether there is a shorter path from the source node to a given node. For this stage, the kernel function - **Algorithm 3.4** - is called.

By leveraging the parallel processing capabilities of GPUs, both Dijkstra's and Bellman-Ford algorithms can be significantly accelerated, making them suitable for large-scale network applications.

---



---

### Algorithm 3. GPU implementation of Bellman - Ford algorithm

---



---

Input: an oriented graph  $G = (V, E)$

Output: MLP in G

```

<<<initialiation>>>(dist)
steps = 0
Repeat
distaux = dist,  $a_r \in E$ 
    <<< Bellman - Ford kernel >>> (G, dist, distaux)
    steps = steps + 1
until distaux = dist or steps = n - 1

```

---

Each kernel (**Algorithm 3.4**) executes one GPU thread for each node  $v$  with index  $id$ , calculating the minimum distance. For this, the previously calculated shortest paths for the predecessors of the nodes are used. If a new, shorter path is found for  $v$ , the distance is updated for node  $v$ . Thus, in each iteration, a new distance vector is calculated, which replaces the old distance vector at the end of the iteration. The algorithm stops when the two distance vectors are the same or if a negative cost cycle is found.

---



---

### Algorithm 4. Bellman-Ford kernel

---



---

Input:

- directed graph  $G = (V, A)$ ,
- dist - distances vector
- dist<sub>aux</sub> - auxiliar distances vector

$id = threadId$

//find the shortest distance from source to id node

$min = INF$

**For** all predecessors  $i$  of  $tid$  **do**

**If**  $w[i, tid] + dist_{aux}[i] < min$  **then**

$min = w[i, tid] + dist_{aux}[i]$

**End if**

**End for**

**If**  $min < dist_{aux}[id]$  **then**

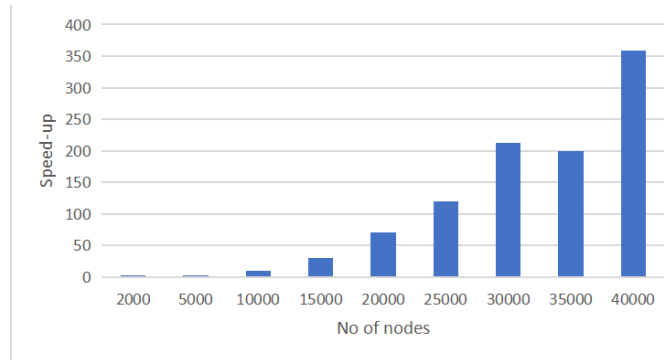
$dist[id] = min$

**End if**

---

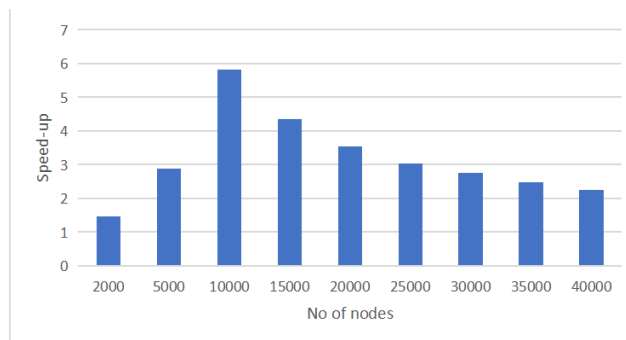
### 3.4. Results and Discussions

To test the MLPP algorithms, random networks were generated using the method described in (Deaconu and Spridon, 2021). The tests were conducted on an Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz 2.59 system, equipped with an NVIDIA GeForce RTX 2060 GPU and running Windows 10. The random networks analyzed had between 2,000 and 50,000 nodes and a varying number of arcs, generated using Algorithm 2.6. Execution times for small networks are very short, and GPU programming is not necessary. As the number of nodes increases, the execution speed increases up to 390 times for networks with 40,000 nodes and up to 326M arcs, as shown in **Figure 3.1**.



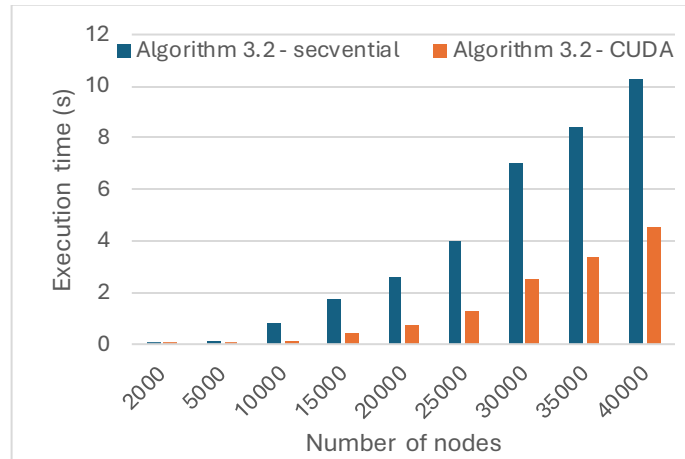
**Figure 0.1** Speed-up on GPU for Algorithm 3.1 on dense networks with varying numbers of nodes (Deaconu, Spridon and Ciupala 2023)

In the case of implementing **Algorithm 3.2**, based on the Bellman-Ford algorithm, the execution time increases with the number of nodes and the density of the network arcs. The highest speed-up on the GPU is 5.8, achieved for a network with 10,000 nodes and 24M arcs (**Figure 3.2**).



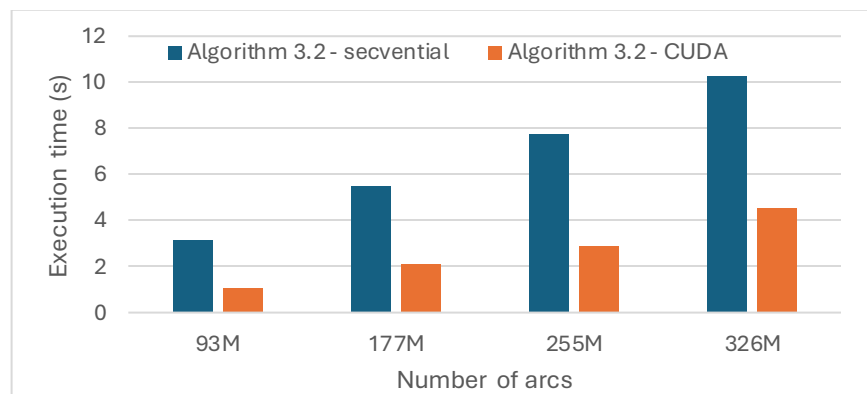
**Figure 3.2** Speed-up on GPU for Algorithm 3.2 on dense networks with varying numbers of nodes (Deaconu, Spridon and Ciupala 2023)

Overall, the execution time on the GPU is significantly lower than on the CPU for all network sizes. This highlights the benefits of CUDA parallelization. Although the GPU execution time increases with the network size, it does so at a slower rate than the CPU execution time (**Figure 3.3**).



**Figure 3.3** Execution time for **Algorithm 3.2** on dense networks

Additionally, as network density increases, so does the execution time for the sequential implementation, whereas for the parallel implementation, the increase in execution time with network density is slower (**Figure 3.4**).



**Figure 3.4** Execution time evolution for **Algorithm 3.2** as a function of network density

The speed-up values are significant, indicating the benefits of CUDA parallelization as networks become larger. The general interpretation of the results shows that CUDA parallelization brings substantial performance improvements for the Bellman-Ford algorithm, especially for large networks.

In conclusion, the results demonstrate increased execution speed when using GPU programming for **Algorithm 3.1** on large and dense networks. Although not as significant, an improvement in execution time was also achieved for **Algorithm 3.2**, using the Bellman-Ford algorithm in the GPU-based implementation.

Comparing the results of the two algorithms, we can observe differences and similarities in their performance. Both algorithms exhibit significant speed-up with CUDA parallelization. Both algorithms benefit from shorter execution times on the GPU compared to the CPU for networks of varying sizes. However, differences are noticeable in how execution time varies with network size for each algorithm. For example, the execution time for **Algorithm 3.1** on the GPU may increase more rapidly with network size, while the Bellman-Ford algorithm seems to have a slower increase (**Figure**



3.3). For both algorithms, the benefits of GPU parallelization become more evident as the problem size grows. The relative performance of the two algorithms may vary depending on network characteristics and other application-specific considerations. The Dijkstra algorithm is known for its lower complexity compared to Bellman-Ford, which may influence relative performance based on specific network features.

## Chapter 4. Determining the Minimum Loss Flow in a Generalized Network

In this chapter, I present a possible application of algorithms for finding the minimum loss path to determine the maximum flow in a generalized network (with losses/gains on the edges). The Ford-Fulkerson algorithm has been adapted to sequentially find s-t paths with minimum loss. I describe two possible implementations of the algorithm: sequential and using GPU parallelization. Multiple tests were conducted for both implementations, comparing execution times. These results were accepted for presentation at the 21st International Conference on Applied Computing (AC 2024) (Spridon, Deaconu, and Popa, et al. 2024).

### 4.1. The Traditional Maximum Flow Problem

In the maximum flow problem, the objective is to send as much flow as possible between two nodes, respecting the capacity limits of the edges. An instance of the maximum flow problem is an antisymmetric network  $G = (V, E, s, t, c)$ , where  $s \in V$  is the source node,  $t \in V$  is the sink node, and  $c$  is a capacity function.

**Theorem 4.1** A flow is maximum if and only if there are no augmenting paths in the residual network.

The proof of Theorem 4.1 can be found in the work of Ford and Fulkerson (1956) .

A residual network is a network  $G_f = (V, E, c_f)$ , where  $c_f: E \rightarrow \mathbb{R}$ ,  $c_f(u, v) = c(u, v) - f(u, v)$ , is the residual capacity function. For example, if  $c(u, v) = 40$ ,  $c(v, u) = 0$ , and  $f(u, v) = -f(v, u) = 29$ , then the arc  $(u, v)$  has  $40 - 29 = 11$  units of residual capacity, and the arc  $(v, u)$  has  $0 - (-29) = 29$  units of residual capacity.

In short, the maximum flow problem is a classic optimization problem in graph theory, involving finding the maximum amount of flow that can be sent through a network of pipes, channels, or other pathways, subject to capacity constraints. The problem can model a wide range of real-world situations, such as transportation systems, communication networks, or resource allocation.

A common approach for solving the maximum flow problem is the Ford-Fulkerson algorithm (Ford and Fulkerson 1956), which is based on the concept of augmenting paths. This algorithm has as input parameters a network,  $G = (V, E, c, s, t)$ , with source node,  $s$ , and sink node,  $t$ , and as output parameter is  $f$ , the maximum flow that can be admitted through the network  $G$ .

## 4.2. The Generalized Maximum Flow Problem

The generalized maximum flow problem extends the traditional maximum flow problem by allowing the flow to change while being transmitted through the network. As before, each edge  $(u, v)$  has a capacity  $c(u, v)$  that limits the amount of flow sent through that edge. In addition, each edge  $(u, v)$  is associated with a positive coefficient  $\alpha(u, v) > 0$ , called the gain/loss factor. The gain/loss factor is a function  $\alpha: E \rightarrow R_{>0}$ . For each unit of flow entering edge  $(u, v)$  from node  $u$ , only  $\alpha(u, v)$  units reach node  $v$ . An edge with losses is one where  $\alpha < 1$ , and an edge where there is a gain has  $\alpha > 1$ . Without loss of generality, we assume that the gain/loss function is symmetric, i.e.,  $\alpha(u, v) = 1/\alpha(v, u)$ . If this assumption is not satisfied, we can add the symmetric edge and assign it a zero capacity (Wayne 1999).

To solve the generalized maximum flow problem for determining the minimum loss flow, we propose an adaptation of the Ford-Fulkerson algorithm (**Algorithm 4.1**) so that, at each iteration, the minimum loss path algorithm (**Algorithm 3.2**) is applied to find a new path in the residual network. The choice of **Algorithm 3.2** is explained by the fact that, both in the case of networks with only losses on edges and in the case of networks with losses or gains on edges, the resulting residual network contains both types of edges due to the way the loss factor is calculated in that network. In other words, if in the initial network an edge  $(u, v)$  has a gain/loss factor  $\alpha(u, v)$ , then the gain/loss factor in the residual network is  $\alpha_f(u, v) = \alpha(u, v)$  on the direct edge, and on the reverse edge, we have  $\alpha_f(v, u) = \frac{1}{\alpha(u, v)}$ .

---

### Algorithm 4.1 Adaptation of the Ford-Fulkerson Algorithm for Determining Maximum Flow in a Generalized Network

---

Input:

- Network  $G = (V, E, s, t, c, \alpha)$

Output:

- $f$  – minimal loss flow in  $G$

Initialize a feasible flow  $f = 0$

Initialize the residual network  $G_f = G$

**Foreach**  $(u, v) \in E$  **do**

$$\alpha_f(u, v) = \alpha(u, v)$$

$$\alpha_f(v, u) = \frac{1}{\alpha(u, v)}$$

**End foreach**

**While** there exists an augmenting path from  $s$  to  $t$  in  $G_f$  **do**

Find an augmenting path  $P$  in  $G_f$  using **Algorithm 3.2**

Update the residual network  $G_f$  using **Algorithm 4.2**

**End while**

---

To update the residual network,  $G_f$ , along the path,  $P$ , found, **Algorithm 3.2** is used. This algorithm involves determining the maximum feasible flow on the augmenting path and then updating the residual capacities and gain/loss factors in the residual network to reflect this flow. Thus, the residual network is prepared for subsequent iterations of the flow algorithm.

The proposed algorithm has two stages:

1. Determination of feasible flow on the augmenting path  $P$ :

- Initialize flow  $f$  with the residual capacity of the first arc  $(s, u)$  in  $P$ .
- For each arc  $(u, v)$  in path  $P$  while node  $u$  is not the destination  $t$ :
  - Update flow  $f$  with the minimum of the current flow  $f \cdot \alpha_f(u, v)$  and the residual capacity,  $c_f(u, v) \cdot \alpha_f(u, v)$ .
  - Go to the next node  $u = v$  on path  $P$ .

2. Update residual network,  $G_f$  :

- Start from the sink node,  $v = t$ .
- For each arc  $(u, v)$  on path  $P$  while node  $v$  is the source  $s$ :
  - Adjust flow  $f$  according to the gain/loss factor  $\alpha_f(u, v)$ .
  - On arc  $(u, v)$ , update the residual capacity  $c_f(u, v) = c_f(u, v) - f/\alpha_f(u, v)$ , and for the reverse arc, add the flow to its residual capacity  $c_f(v, u) = c_f(v, u) + f$ .
  - Advance on path  $P$  towards the source, to the previous node  $v = u$ .

---

#### Algorithm 4.2. Updating the Residual Network after Finding a New Augmenting Path in $G_f$

---

Input:

- Residual network  $G_f = (V, E, s, t, c_f, \alpha_f)$
- Augmenting path from  $s$  to  $t$  found,  $P$  in  $G_f$

Output:

- Updated residual network,  $G_f$
- Feasible flow,  $f$

// Determine the flow value at node  $t$  on path  $P$

Consider a flow,  $f = c_f(s, u)$ , where  $(s, u) \in P$  is the first arc in  $P$

$u = s$

**while**  $u \neq t$  **do**

Consider the arc  $(u, v) \in P$

$f = \min\{f \cdot \alpha_f(u, v), c_f(u, v) \cdot \alpha_f(u, v)\}$

$u = v$

**End while**

// Update  $G_f$

$v = t$

**while**  $v \neq s$  **do**

Consider the arc  $(u, v) \in P$

$c_f(u, v) = c_f(u, v) - f/\alpha_f(u, v)$

$c_f(v, u) = c_f(v, u) + f$

$f = f/\alpha_f(u, v)$

$v = u$

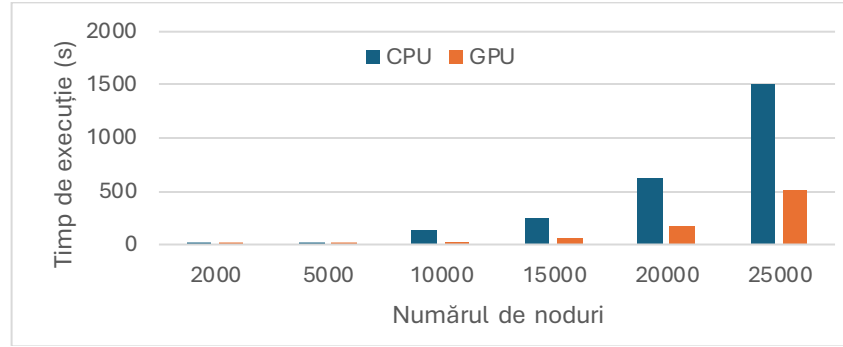
**End while**

---

### 4.3. Results and Discussions

When implementing the proposed algorithm, the acceleration results when using GPU programming are similar to those obtained for **Algorithm 3.2**, as the efficiency can be achieved by using the parallel Bellman-Ford algorithm for each determination of a minimum loss path. Thus, as can be seen in Algorithm 4.1, its complexity is given by the complexity of **Algorithm 3.2** multiplied by the number of iterations in which a path augmentation is determined.

GPU efficiency initially increases with the complexity of the problem but starts to decrease after a certain point, possibly due to GPU resource saturation. The speed-up varies between 1.04 and 5.72, with maximum values for medium-sized problems. GPU performance is significantly superior to CPU for most tested configurations, especially for medium and large graph sizes.



**Figure 4.4** Execution Times for Algorithm 4.1 in Dense Networks

As shown in **Figure 4.1**, the execution time on the GPU is significantly shorter than on the CPU for all analyzed dense networks, regardless of the number of nodes. GPU efficiency is particularly evident for large, dense networks. Thus, the execution time on the GPU increases more slowly compared to the CPU as the number of nodes grows. The speed-up ratio decreases as the number of nodes increases. For smaller networks (2000-10000 nodes), the GPU offers considerable speed-up (2.7-5.7 times). For larger networks (15000-25000 nodes), this ratio slightly decreases but remains significant (2.98-4.45 times).

## Chapter 5. Fast GPU Interpolation for Map Generation

This chapter builds upon the works (**Spridon**, Deaconu, and Ciupala, ICCSA 2023) and (Ciupala, Deaconu, and **Spridon** 2021), to which I am a co-author. In this chapter, I present GPU methods for generating pollution and geomagnetic maps using interpolation techniques, starting from measurements taken at various points within a specific geographical area. I have conducted accuracy analyses of the generated maps and the efficiency of the used GPU methods, with the results presented below.

### 5.1. Two-Dimensional Interpolation Methods

The general formulation of the spatial interpolation problem can be defined as follows:

Given  $n$  values of a studied phenomenon  $V(j)$ , with  $j = \overline{1, n}$ , measured at discrete coordinate positions  $r_j = (x_j, y_j)$ , in a two-dimensional space, the goal is to find a function  $F(r)$  that satisfies the conditions:

$$F(r_j) = V_j, \forall j = \overline{1, n} \quad (5.1)$$

Since there are infinitely many functions that satisfy this requirement, additional conditions must be imposed, which define the nature of different interpolation techniques. Typical examples include conditions based on geostatistical concepts (kriging), localization (nearest neighbor and finite element methods), smoothness, and splines or ad-hoc functional forms (polynomials, multi-quadrics). The choice of additional condition depends on the nature of the modeled phenomenon and the type of application.

Several interpolation methods are used to generate maps in fields such as cartography, geography, and spatial data analysis. Below, I will describe two of the most recent methods used for this purpose.

- The *IDW (Inverse Distance Weighting)* method assumes that the estimated value is a function of the distance between the estimation point and the sample locations, such that measured values closer to the point of estimation have a greater influence on the estimated value than those further away.
- *Kriging* involves estimating the unknown value  $z(u)$  at a specific location  $u$  based on a weighted average of observed values  $V(r_i)$  at nearby sample points  $r_i$ . The weights are chosen to minimize the estimation error and are determined based on the spatial correlation structure of the data. Kriging weights are obtained by solving a system of linear equations that express the spatial autocovariance function of the data.

## 5.2. Accelerating Interpolation Methods Using CUDA

The implemented algorithms were tested on a system with an Intel(R) Core(TM) i7-10750H @ 2.60GHz processor, 16.0 GB RAM, NVIDIA GeForce RTX 2060 GPU, and Windows 10 Pro operating system. These interpolation methods have been accelerated using CUDA programming to generate high-resolution maps in real-time. This tool could be used, for example, for monitoring geomagnetic changes over large areas to identify changes that may occur in Earth's structure or for identifying regions with specific magnetic properties or real-time monitoring of pollution maps in various areas.

The pseudocode for the IDW algorithm is presented below (Ciupala, Deaconu, and Spridon).

---



---

### Algorithm 5.1 IDW algorithm

---



---

**Input:**  $p, x_{min}, x_{max}, y_{min}, y_{max}, n, m;$

/\* Determine resolution in the x and y directions \*/  $dx = \frac{x_{max} - x_{min}}{n}$

$dy = \frac{y_{max} - y_{min}}{m}$

/\* Compute the estimated values at each point of the grid \*/

$y = y_{min}$

**For**  $i = 1, n$  **do**

$x = x_{min}$

**Pentru**  $j = 1, m$  **execută**

$g_{ij} = v(x, y)$

$x = x + dx$

**End for**

$y = y + dy$

**End for**

---

where  $g_{ij}$  are the estimated values on a 2D  $m \times n$  grid,  $m, n \in \mathbb{N}^*$  for a rectangular region defined by the coordinates  $x_{min}, x_{max}, y_{min}, y_{max} \in \mathbb{R}$ , ( $x_{min} < x_{max}, y_{min} < y_{max}$ ). Thus, **Algorithm 5.1** creates a 2D grid over a surface bounded by the coordinates  $x_{min}, x_{max}, y_{min}, y_{max}$ .  $dx$  and  $dy$  are calculated to determine the distance between grid points along the  $x$  and  $y$  axes, respectively. subsequently, all grid points are processed to compute the value  $g_{ij}$  at each coordinate  $(x, y)$  using a weighted average  $V(x, y)$ . The distance between two grid points is calculated using the formula for the distance on Earth between two points with given GPS coordinates.

To use CUDA for IDW, we first need to parallelize the algorithm. In IDW, we must calculate the distance between the estimation points and each of the sample points. This distance calculation can be parallelized by assigning each GPU thread to a single grid point and calculating distances to all sample points.

After calculating the distances, we compute the weights for each sample point based on the distance to the estimation point. This weight calculation can also be parallelized by assigning each GPU thread to a single sample point and computing its weight for all points where the value is to be estimated.

Finally, we can use the calculated weights to interpolate values at the grid points. This interpolation step can also be parallelized by assigning each GPU thread to a single estimation point and calculating its value based on the weighted average of the sample points' values.

The kriging interpolation algorithm was implemented following 4 steps (**Algorithm 5.2**).

---



---

### Algorithm 5.2 Kriging algorithm

---



---

Input:  $x_{min}, x_{max}, y_{min}, y_{max}, n, m, v;$

Calculation of Semivariance Points

Calculation of Semivariance Coefficients Using the Least Squares Method

Calculation of Interpolation Weights

Calculation of Interpolated Values

---

For parallelizing the kriging algorithm using CUDA, several steps are required: calculate the variogram, compute the kriging matrix, and calculate kriging weights.

The calculation of the variogram involves determining the semivariance between all pairs of sample points. This step can be parallelized by assigning each GPU thread to a single pair of sample points and calculating their semivariance.

The calculation of the kriging matrix involves inverting a matrix that depends on the semivariances between sample points.

Finally, the calculation of kriging weights involves determining the weights for each sample point based on its distance and spatial correlation with the estimation point. This step can be parallelized by assigning each GPU thread to a single estimation point and calculating its weights for all sample points.

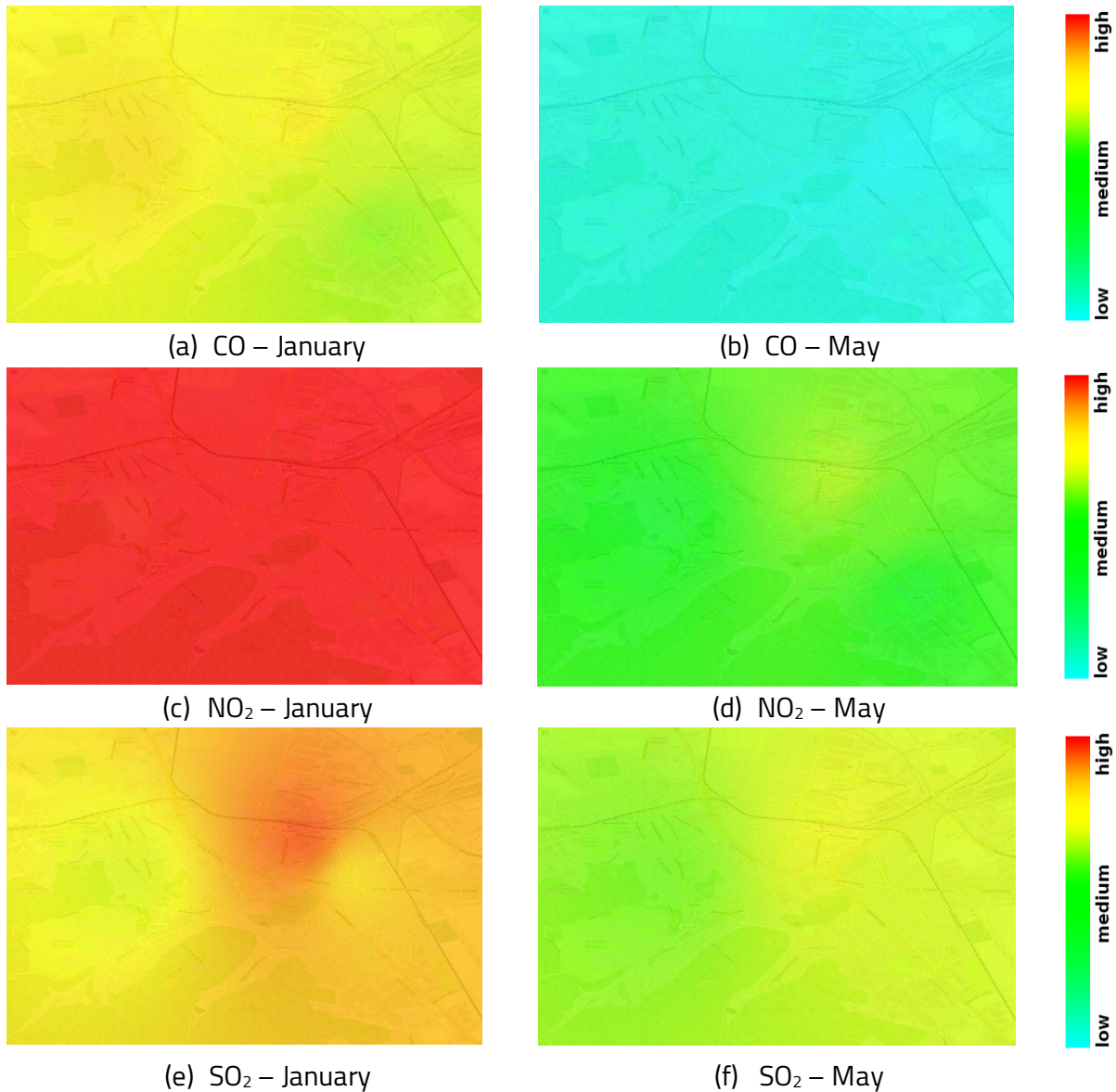
### ***5.3. Study of Air Pollution Maps for Braşov During the Pandemic***

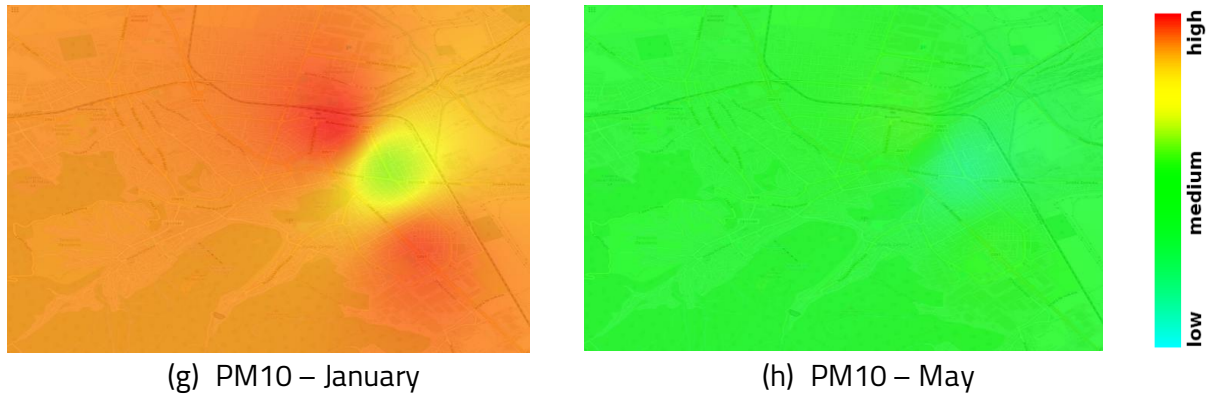
We used the IDW method to create pollution maps (grids) for the urban area of Braşov and to draw conclusions about pollution for the year 2020. We also compared air quality during the lockdown period (when most economic and social activities were halted) due to the Covid-19 pandemic and the period when the economy was restarted. For this study, concentrations of carbon monoxide (CO), sulfur dioxide (SO<sub>2</sub>), nitrogen dioxide (NO<sub>2</sub>), and particulate matter (PM<sub>10</sub>) were considered. Data for the four stations reporting hourly pollution in Braşov were downloaded from the National Air Quality Monitoring Network 2021 for the first half of 2020 for CO, PM<sub>10</sub>, SO<sub>2</sub>, and NO<sub>2</sub>.

Using the IDW algorithm, we generated maps of hourly pollutant concentrations, 24-hour average maps for each pollutant, monthly average concentration maps, and average concentration

maps for each day of the week to see how pollution differs on weekdays compared to weekends. We also compared weekly statistics graphically and tracked monthly pollution trends. This was done separately for each of the four stations and averaged for all stations.

For each pollutant, we created two maps to compare the average concentration in January (before the lockdown) and May (the last month of lockdown) (see **Figure 5.1**). Comparing the two images, it is evident that air quality improved significantly for each pollutant due to reduced industrial activity and the lower number of vehicles operating during that period.





**Figure 5.1** Comparison of the average concentrations of major pollutants for January 2020 (a, c, e, g) and May 2020 (b, d, f, h)

The experimental results presented in Table 5.1 showed that CUDA-based implementations running on GPUs led to increased execution speed depending on the image resolution. IDW interpolation was used to obtain images ranging in size from 150 x 100 to 4800 x 3200. The experiments demonstrated that for small images (150 x 100 and 300 x 200), the CPU time was better. For large images, GPU acceleration was consistent, up to 19 times faster.

**Table 5.1** GPU Execution Time Study

Image size	CPU Execution Time (s)	GPU Execution Time (s)	Speed-up
100 x 150	0.017	0.031	0.55
300 x 200	0.065	0.071	0.92
600 x 400	0.268	0.101	2.65
1200 x 800	1.090	0.167	6.52
2400 x 1600	4.415	0.322	13.71
4800 x 3200	17,913	0.942	19.02

#### 5.4. Study of Geomagnetic Maps of Romania

Geomagnetic data and maps are essential tools for understanding the Earth's magnetic field and its various applications. Geomagnetic data provide insights into the structure and dynamics of the Earth's interior, while geomagnetic maps are used for navigation, geological mapping, and scientific research. These maps and data have practical applications in industry and commercial enterprises, particularly in mineral exploration, energy development, and navigation.

#### 5.5. CUDA Methods for Generating Geomagnetic Maps

Geomagnetic data for generating the geomagnetic map of Romania, using IDW and kriging interpolation methods, were obtained from Romanian geomagnetic stations and through the Physics Toolbox Sensor Suite application at over 1300 GPS positions spread across the country. The data were collected through the Citizen Science initiative of the European Researchers' Night 2018-2019 Handle with Science project, funded by H2020, AG no. 818795/2018.

The studied region lies between 21° E and 29° E longitude and between 41° N and 49° N latitude. The grids obtained have resolutions of 400 x 400, 800 x 800, 1200 x 1200, and 1600 x 1600, so each grid point is approximately 2 km, 1 km, 0.75 km, and 0.5 km, respectively. **Figures 5.2** and **5.3** show geomagnetic maps with 1 km resolution for the Romania region obtained using IDW and kriging interpolation, respectively.



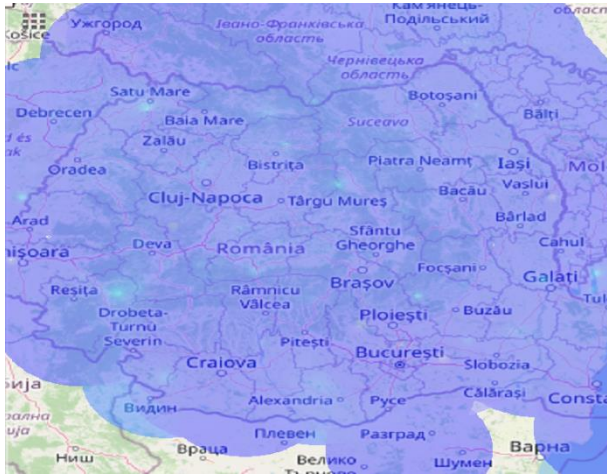


Figure 5.2 Geomagnetic map obtained using IDW

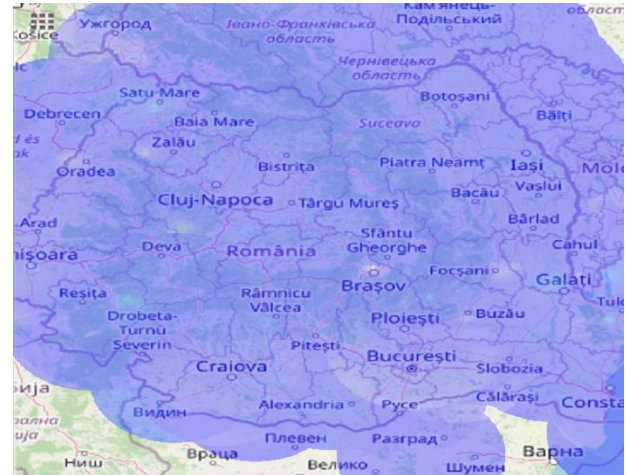


Figure 5.3 Geomagnetic map obtained using kriging interpolation

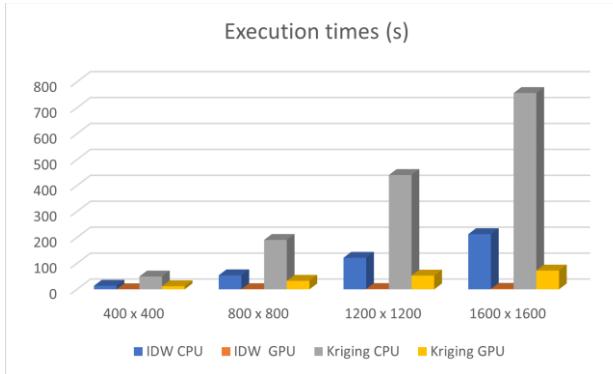
The results show better accuracy for the kriging interpolation method across all studied resolutions. For example, while the median error value for IDW ranges between 4.476 and 4.895  $\mu\text{T}$  depending on resolution, for kriging, this value ranges between 2.871  $\mu\text{T}$  and 3.687  $\mu\text{T}$ . Furthermore, Figure 5.4 shows lower average error values for the geomagnetic field with the kriging method.



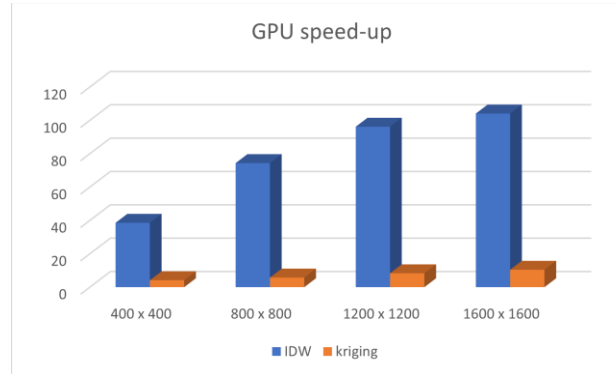
Figure 5.4 Comparison of average error for the geomagnetic field obtained via IDW and Kriging

In other words, comparing the results, we can see that the kriging interpolation method has lower error values for all analyzed errors compared to IDW, indicating better performance for this method in geomagnetic data interpolation. Additionally, in general, as the grid resolution increases, the error values decrease, indicating improved interpolation accuracy with higher resolution.

The complex calculations involved in the kriging method lead to increased execution time for all resolutions compared to IDW, as shown in Figure 5.5. The speed achieved for the IDW implementation is very high and increases with the grid resolution, up to 104 times for the 1600x1600 grid. Although kriging speed is not as high as IDW (Figure 5.6), for the highest resolution studied, GPU execution time decreased by 10 times compared to CPU. Thus, it is observed that execution time for both IDW and kriging methods is significantly lower for GPU implementations.



**Figure 5.5** Execution times for IDW and Kriging on CPU and GPU



**Figure 5.6** Speed-up for IDW and Kriging

Execution time on the GPU is significantly lower than on the CPU for both interpolation methods and all grid sizes. This indicates that parallelizing interpolation algorithms using CUDA leads to significant acceleration of the interpolation process. Specifically, speed-up increases with the grid size, showing that the benefits of parallelization are more pronounced for larger datasets. Thus, implementing interpolation algorithms on the GPU using CUDA can be an efficient choice for improving performance and execution time. Comparing the speed-up for the two methods, it is observed that, generally, the speed-up for Kriging is lower than for IDW for any grid size analyzed. This suggests that parallelizing the Kriging interpolation algorithm on the GPU using CUDA brings smaller benefits compared to IDW. However, both methods can clearly benefit from GPU acceleration, and the difference in speed-up is influenced by the specific nature of the algorithms and their parallelization. Nevertheless, considering both execution performance and result accuracy, it can be concluded that although IDW provides a higher speed-up and shorter execution time, Kriging is a better option when aiming for high-precision results despite a longer execution time.

## Chapter 6. Conclusions and Future Perspectives

In this work, **Chapter 1** presents several advantages and disadvantages of GPU programming and reviewed some of the most important applications of GPU programming. This information was published in the paper (Spridon, Advances in CUDA for computational physics, 2023).

The following chapters present some of my personal results published in scientific journals or presented at international conferences. **Chapter 2** introduces a fast and reliable algorithm called AGRFA for generating random networks. The resulting networks can be used to test the correctness and efficiency of algorithms developed for network flow problems, such as minimum cost flow, maximum flow, or multi-commodity flow problems. The CUDA-parallelized version of AGRFA has proven to be up to 19 times faster for generating large networks. With subsequent developments, other specific network problems where AGRFA can be adapted could be identified. These results were published as a co-author in the paper (Deaconu and Spridon 2021).

**Chapter 3** introduces and solves a practical problem called the minimum loss path problem or maximum delivery rate path. This problem involves finding a path from a source node to a given sink node in a generalized network, where each arc has an associated gain/loss factor, such that the loss is minimized among all s-t paths. The results show high speed when using GPU programming for

Algorithm 3.1 on large and dense networks. Execution time improvement was also achieved for Algorithm 3.2 using the Bellman-Ford algorithm in the GPU-based implementation. The results were presented in the paper (Deaconu, **Spridon**, and Ciupala 2023).

**Chapter 4** presents an application for determining the minimum loss path. Thus, the MLPP algorithm is used in a generalized network to determine the minimum flow. An adaptation of the Ford-Fulkerson algorithm is proposed, where in each iteration the path with minimum loss is sought. This results in the minimum loss flow in the network.

In **Chapter 5**, the generation of georeferenced maps using two-dimensional interpolation methods based on measured values at discrete points in a given geographical area is described. Thus, pollution maps of Braşov during the COVID-19 lockdown were studied. The maps were obtained using the IDW interpolation method, and for high resolutions, CUDA was used, resulting in significant speed-up in execution time. Additionally, geomagnetic maps of Romania were studied using IDW and kriging interpolation methods, investigating both the accuracy of the obtained maps and their generation speed. The estimation errors in the geomagnetic maps are lower for the kriging interpolation method, and execution speed was shown to be improved using GPU programming with CUDA. The works underpinning this chapter are (Ciupala, Deaconu, and **Spridon** 2021) and (**Spridon**, Deaconu, and Ciupala, ICCSA 2023).

As future research perspectives, in the field of two-dimensional interpolation, I aim to study the GPU parallelization of other interpolation algorithms on irregular, non-uniformly distributed datasets and obtaining high-resolution maps. Additionally, I want to develop hybrid models that combine the processing power of GPUs with CPU parallelization methods to increase the execution speed of algorithms when applied to large networks. This requires studying and evaluating the performance of parallel algorithms in the context of graph theory on GPUs, and identifying limitations and optimizing code to fully leverage CUDA architecture. Moreover, I plan to use GPU methods to accelerate the computation and solve complex problems in computational physics, particularly focusing on accelerating simulations modeling energy transport, particles, and interactions within plasmas, and developing GPU algorithms to analyze complex phenomena such as turbulent transport in plasmas.

## Published Works in the Field of the Thesis

### Works Published in ISI Impact Factor Journals:

1. Deaconu, A.M, **Spridon. D.**, „Adaptation of Random Binomial Graphs for Testing Network.” *Mathematics* 9 (2021): 1716.

### Works Published in Scopus-Indexed Journals:

1. **Spridon, D.** 2023. "Advances in CUDA for computational physics", Bulletin of the Transilvania University of Braşov. Series III: Mathematics and Computer Science, 65 (2): 227-236.
2. Ciupala, L., Deaconu, A., **Spridon. D.**, 2021. "IDW map builder and statistics of air pollution in Brasov.", *Bulletin of the Transilvania University of Braşov. Series III: Mathematics and Computer Science*, 63(1), 247-256.

### Papers Presented and Published in Proceedings of International Conferences Indexed by CORE:

1. **Spridon, D.**, Deaconu, A. M., Ciupala, L. 2023. "Fast CUDA Geomagnetic Map Builder." Lecture Notes in Computer Science, International Conference on Computational Science and Its Applications, 126 -138, Athens: Springer.
2. Deaconu, A.M., **Spridon, D.E.** , Ciupala, L. 2023. "Finding minimum loss path in big networks." *International Symposium on Parallel and Distributed Computing*. Bucharest: IEEE. 39-44.
3. **Spridon, D.**, Deaconu, A. M., Popa, I., Tayyebi, J., New approach for the generalized maximum flow problem, accepted to 21st International Conference on Applied Computing, Zagreb, Croatia, 2024

## Selective bibliography

Ahuja, R.K., T.L. Magnanti, and J.B. Orlin. 1993. *Network Flows: Theory, Algorithms, and Applications*; NJ, USA: Prentice Hall: Englewood Cliffs.

Baji, T. 2018. „Evolution of the GPU Device widely used in AI and Massive Parallel Processing.” *IEEE 2nd Electron Devices Technology and Manufacturing Conference*. Kobe: IEEE. 7-9.

Bellman, R. 1958. „On a routing problem.” *Quarterly of Applied Mathematics* 87-90.

Ciupala, L., A. Deaconu, and D. **Spridon**. 2021. „IDW map builder and statistics of air pollution in Brasov.” *Bulletin of the Transilvania University of Brasov. Series III: Mathematics and Computer Science*, 247-256.

Deaconu, A. 2006. „A Cardinality Inverse Maximum Flow Problem.” *Scientific Annals of Cuza University* 16: 51-62.

Deaconu, A., and E. Ciurea. 2012. „The inverse maximum flow problem under Lk norms.” *Carpathian Journal of Mathematics* 28: 59-66.

Deaconu, A., and L. Ciupala. 2020. „Inverse Minimum Cut Problem with Lower and Upper Bounds.” *Mathematics* 8: 1494.

Deaconu, A.M, and D. **Spridon**. 2021. „Adaptation of Random Binomial Graphs for Testing Network.” *Mathematics* 9: 1716.

Deaconu, A.M., D.E. **Spridon**, and L. Ciupala. 2023. „Finding minimum loss path in big networks.” *International Symposium on Parallel and Distributed Computing*. Bucharest: IEEE. 39-44.

Dijkstra, E.W. 1959. „A note on two problems in connexion with graphs.” *Numerische Mathematik* 269-271.

Durstenfeld, R. 1964. „Algorithm 235: Random permutation.” *Communications. ACM* 7: 420.

Ford, L.R., and D. R. Fulkerson. 1956. „Maximal flow through a network.” *Canadian Journal of Mathematics* 399-404.

Fredman, M.L., and R.E. Tarjan. 1984. „Fibonacci heaps and their uses in improved network optimization algorithms.” *25th Annual Symposium on Foundations of Computer Science*. IEEE. 338-346.

Harish, P., and P. J. Narayanan. 2007. „Accelerating Large Graph Algorithms on the GPU using CUDA.” *Lecture Notes in Computer Science*.

Huang, J. 2023. *NVIDIA*. October. [https://s201.q4cdn.com/141608511/files/doc\\_presentations/2023/Oct/01/ndr\\_presentation\\_oct\\_2023\\_final.pdf](https://s201.q4cdn.com/141608511/files/doc_presentations/2023/Oct/01/ndr_presentation_oct_2023_final.pdf).

Marinescu, C., A. Deaconu, E. Ciurea, and D. Marinescu. 2010. „From Microgrids to Smart Grids: Modeling and Simulating using Graphs. Part II Optimization of Reactive Power Flow.” *12th International Conference on Optimization of Electrical and Electronic Equipment*. Brasov. 1251-1256.

—. 2010. „From microgrids to smart grids: Modeling and simulating using graphs. Part I active power flow.” *12th International Conference on Optimization of Electrical and Electronic Equipment* Brasov. 1245-1250.

Martin, P., R Torres, and A. Gavilanes. 2009. „CUDA solutions for the SSSP.” *Computational Science – ICCS*. Springer Berlin / Heidelberg. 904-913.

Ortega-Arranz, H., Y. Torres, D. R. Llanos, and A. Gonzalez-Escribano. 2013. „A new GPU-based approach to the Shortest Path problem.” *HPCS*. Helsinki. 505-511.

Reţeaua Naţională de Monitorizare a Calităţii Aerului. 2021. <https://www.calitateaer.ro>.  
<https://www.calitateaer.ro>.

Schrijver, A. 2012. „On the history of the shortest path problem.” *Documenta Mathematica, Extra Volume ISMP155*–167.

**Spridon, D.** 2023. „Advances in CUDA for computational physics.” *Bulletin of the Transilvania University of Brasov*. Series III: Mathematics and Computer Science, 3(65) (2): 227-236.

**Spridon, D., A. M. Deaconu, I. Popa, and J. Tayyebi.** „New approach for the generalized maximum flow problem.” accepted to 21st International Conference on Applied Computing. Zagreb, Croatia, 2024.

**Spridon, D., A. M. Deaconu, and L. Ciupala.** 2023. „Fast CUDA Geomagnetic Map Builder.” *Lecture Notes in Computer Science*. Athens: Springer.

Surve, G. G., and M. A. Shah. 2017. „Parallel implementation of Bellman-Ford algorithm using CUDA architecture.” *ICECA*. Coimbatore.

Sven, O.K., and C. Zeck. 2013. „Generalized max flow in series-parallel graphs.” *Discrete Optimization* 10: 155–162.

Tayyebi, J., and A.M. Deaconu. 2019. „Inverse Generalized Maximum Flow Problems.” *Mathematics* 899.

Wayne, K.D. 1999. <https://www.cs.princeton.edu/~wayne/papers/thesis.pdf>. January.  
<https://www.cs.princeton.edu/~wayne/papers/thesis.pdf>.