Universitatea Transilvania din Brașov

ȘCOALA DOCTORALĂ INTERDISCIPLINARĂ
Facultatea de Matematică și Informatică

Delia Elena CUZA (căs. SPRIDON)

Metode GPU pentru creșterea performanței computaționale în teoria grafurilor și generarea hărților

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

REZUMAT

Conducător ştiințific
Prof. dr. Marius Sabin TĂBÎRCĂ

BRAȘOV, 2024

Cuprins

INTRODUCERE

În această lucrare, am investigat utilizarea tehnologiei CUDA (Compute Unified Device Architecture) pentru optimizarea și eficientizarea unor algoritmi specifici în diverse domenii. Am structurat principalele contribuții și rezultate proprii, fiecare aducând îmbunătățiri semnificative în domeniul său de aplicare prin utilizarea capacităților de calcul paralel oferite de GPU-urile (Graphics Processing Unit) NVIDIA. Această lucrare are la bază șase lucrări publicate în reviste științifice de prestigiu sau prezentate la conferințe internaționale, toate fiind indexate în baze de date internaționale recunoscute și o lucrare acceptată pentru prezentare și publicare în proceedings-ul unei conferințe indexată CORE.

Prima contribuție majoră a fost dezvoltarea și implementarea algoritmilor noi pentru generarea de rețele aleatorii, care sunt esențiale pentru modelarea și simularea diverselor fenomene naturale și sociale. Rețelele aleatorii sunt utilizate în numeroase aplicații, de la analiza structurii sociale și până la simularea proceselor de difuzie în fizică și chimie. Utilizarea tehnologiei CUDA a permis o accelerare semnificativă a procesului de generare a acestor rețele. În comparație cu metodele tradiționale bazate pe CPU (Central Processing Unit), soluția propusă a redus timpul de execuție prin paralelizarea operațiilor de generare, fapt care s-a tradus într-o creștere semnificativă a performanței și a capacității de a gestiona rețele de mari dimensiuni.

A doua contribuție a fost propunerea și implementarea unor algoritmi pentru determinarea drumurilor cu pierderi minime în rețele generalizate. Acești algoritmi sunt critici în diverse aplicații de optimizare a fluxurilor, precum cele din domeniul logisticii, transportului și rețelelor de telecomunicații. Implementarea lor folosind tehnologia CUDA a permis procesarea paralelă a nodurilor și muchiilor din rețea, ceea ce a dus la o reducere semnificativă a timpului de calcul. În loc să se proceseze fiecare drum în mod secvențial, GPU-urile au permis efectuarea simultană a multiplelor calcule, astfel încât soluția optimă a fost obținută mult mai rapid. Această eficiență crescută a fost demonstrată prin teste pe rețele complexe, unde algoritmii CUDA au redus timpul necesar pentru determinarea drumurilor optime în comparație cu soluțiile tradiționale pe CPU.

Ca aplicație practică a algoritmilor de determinare a drumurilor cu pierderi minime, am propus o soluție pentru rezolvarea problemei minimizării pierderii de flux în rețele. Această problemă este deosebit de relevantă în contextul rețelelor de distribuție de energie, de apă sau alte tipuri de rețele în care pot exista pierderi de-a lungul arcelor. Optimizarea pe GPU folosind CUDA a permis realizarea unor calcule intensive într-un timp mult mai scurt decât abordările clasice. Algoritmii dezvoltați au fost testați pe rețele de dimensiuni mari și au demonstrat o eficiență ridicată în identificarea și minimizarea pierderilor. Rezultatele au arătat că utilizarea GPU-urilor nu doar că accelerează procesul de calcul.

În final, am aplicat metode de interpolare, precum Inverse Distance Weighting (IDW) și kriging, utilizând CUDA pentru a genera hărți de poluare și geomagnetism precise și detaliate într-un timp scurt. Interpolarea este o metodă crucială pentru cartografierea datelor spațiale, folosită în geografie, meteorologie și alte științe ale Pământului. Implementarea acestor metode pe GPU a permis

paralelizarea calculului distanțelor și ponderilor, ceea ce a dus la o accelerare semnificativă a procesului de interpolare. Această accelerare a fost deosebit de utilă în cazul seturilor de date mari și complexe, unde calculele tradiționale ar putea fi mult prea lente.

În concluzie, prin utilizarea tehnologiei CUDA, am reușit să optimizăm și să eficientizăm algoritmi esențiali pentru diverse aplicații, demonstrând că GPU-urile pot aduce îmbunătățiri semnificative în performanța și scalabilitatea acestor algoritmi. Această lucrare subliniază potențialul uriaș al calculului paralel pe GPU în rezolvarea problemelor complexe și deschide noi direcții pentru cercetări viitoare în domeniu. Astfel, cercetările prezentate aici nu doar că sunt fundamentate pe o bază solidă de studii și experimente publicate și validate internațional, dar și demonstrează o aplicabilitate practică extinsă în multiple domenii de interes științific și tehnologic.

Pe scurt teza de față se bazează pe rezultatele obținute și publicate în reviste sau proceedings-urile unor conferințe recunoscute la nivel internațional. Astfel, în domeniul tezei, am publicat:

- 1 articol ISI încadrat în lista A de reviste
- 2 articole într-o revistă indexată Scopus
- 3 articole prezentate și publicate în proceedings-urile unor conferințe clasificate CORE C
- 1 articol acceptat pentru prezentare la o conferință indexată CORE C

În **Tabelul 1** se prezintă încadrarea acestor lucrări conform standardelor de evaluare a tezelor de doctorat în domeniul Informatică, valabile în momentul susținerii tezei.

Mai mult, din punct de vedere al impactului rezultatelor, subliniez faptul că lucrările publicate în domeniul tezei au 11 citări (fără autocitări), dintre care:

- 4 citări sunt în reviste cotate ISI
- 1 citare în revistă indexată Scopus
- 2 citări în proceedings conferințe clasificate CORE C
- 3 citări de categorie D.

Tabel 1 Lucrări publicate și punctajul corespunzător, conform standardelor de evaluare a tezelor de doctorat 2018.10.01-prezent, citări (fără auto-citări)

Nr. crt.	Nr. autori	Titlu articol	Revista / Proceeding	Baza de date internațională	Punctaj [1]	Citări
1.	2	Adaptation of Random Binomial Graphs for Testing Network.	Mathematics	ISI – A	8p	3

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

2.	1	Advances in CUDA for computational physics	Bulletin of the Transilvania University of Brasov. Series III: Mathematics and Computer Science	Scopus	2p	1
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	-
Total					17p	11

În **Tabelul 2** am prezentat îndeplinirea standardelor naționale minimale curente de acordare a titlului de doctor în domeniul Informatică. Astfel, în domeniul tezei, am publicat:

- 1 lucrare într-o revistă cotată ISI
- 2 lucrări într-o revistă indexată Scopus
- 4 lucrări au fost prezentate la conferințe internaționale, dintre care 3 sunt in ISI, CORE C, Scopus, DBLP, IEEE/Springer etc.

Tabel 2 Îndeplinirea standardelor naționale minimale pentru acordarea titlului de doctor – Comisia de Informatică [2]

Nr. crt.	Criteriu	Tip lucrări	Nr lucrări
1.	Publicarea sau acceptarea spre publicare (cu prezentarea dovezii de accept) a minimum un articol în reviste indexate ISI din lista UEFISCDI sau în reviste indexate SCOPUS	ISI	1
		Scopus	2
2.	Participarea și susținerea a minimum două lucrări științifice la conferințe internaționale dovedite prin programul conferinței		4 (dintre care 3 sunt indexate ISI, CORE C, Scopus, DBLP, IEEE/Springer etc.
3.	Conferințele recunoscute sunt cele indexate în următoarele baze de date: SCOPUS, IEEE, ACM, SPRINGER, DBLP, CiteSeerX, Zentralblatt, MathSciNet, COPERNICUS, EBSCO şi ProQuest.		

[2] Standarde naționale minimale pentru acordarea titlului de doctor: https://www.edu.ro/sites/default/files/fisiere%20articole/OMEN%205110-2018.pdf

Capitolul 1. Calcul de înaltă performanță pe plăci grafice

În acest capitol sunt prezentate studii publicate în lucrarea (Spridon, Advances in CUDA for computational physics 2023) care îmi aparține. Acesta face un rezumat al celor mai importante rezultate ale cercetării din ultimii ani în ceea ce privește programarea pe GPU. De asemenea, sunt prezentate comparativ cele mai cunoscute metode de programare pe GPU și sunt subliniate avantajele și limitările programării pe GPU folosind tehnologia CUDA.

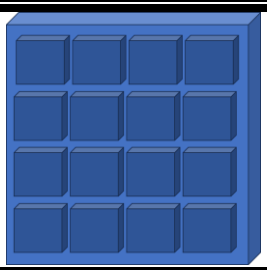
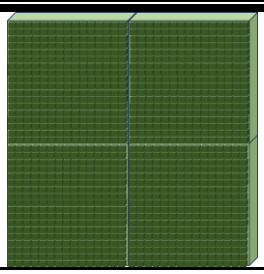
1.1. Generalități

Calculul de înaltă performanță (HPC - High Performance Computing) este un domeniu al informaticii care se concentrează pe utilizarea sistemelor și tehnologiilor pentru a realiza calcule complexe sau intensive din punct de vedere computațional la viteze și eficiență superioare. Acest domeniu se ocupă adesea de rezolvarea problemelor dificile și de manipularea unor cantități masive de date într-un timp cât mai scurt posibil. Una dintre cele mai accesibile metode pentru a realiza acest lucru este utilizarea unităților de procesare grafică (GPU).

În **Tabel 1.1** sunt prezentate comparativ CPU și GPU. Pe scurt, programarea GPU oferă multe beneficii, inclusiv procesare paralelă, eficiență energetică, rentabilitate și flexibilitate. Cu toate acestea, necesită, de asemenea, cunoștințe și experiență specializate, are o suprasarcină suplimentară pentru transferul de date și nu este aplicabilă tuturor tipurilor de aplicații. În plus, câștigurile de performanță ale programării GPU sunt limitate de limitările hardware, dar aplicațiile pe scară largă pot necesita hardware specializat sau mai multe GPU pentru a obține performanțe optime.

Eficiența GPU-ului poate fi direct proporțională cu numărul de nuclee GPU. Datorită acestui fapt, GPU-ul poate beneficia 100% de legea lui Moore sau de creșterea constantă a densității de integrare. Creșterea performanței GPU-ului continuă să țină ritmul de 1,5 ori pe an. În 2017, câștigul de performanță față de CPU era de 10-100 de ori, în funcție de aplicație. Până în anul 2025, se estimează că aceasta va fi de aproape 1.000 de ori. Astfel, dacă în ziua de astăzi pentru CPU legea lui Moore a încetinit, iar unii chiar spun că s-a terminat, creșterea puterii de calcul pe GPU își menține ritmul (Huang 2023).

Tabel 1.1 *Comparație CPU / GPU*

CPU	GPU
	
Până la câteva zeci de nuclee, foarte puternice	Până la câteva mii de nuclee, optimizate pentru paralelism

Frecvențe mai mari pentru execuția rapidă a instrucțiunilor	Frecvențe relativ mai mici, dar operații paralele eficiente
Memorie cache mai mare și eficientă pentru sarcini de procesare generală	Memorie cache mai mică, dar optimizată pentru seturi de date mari specifice graficii, în general
Ideal pentru sarcini de calcul single-thread sau multi-thread ușor	Optimizat pentru grafică, procesare paralelă și algoritmi masiv paraleli
Consum mai redus de energie, ideal pentru sisteme portabile	Consum mai mare de energie
De obicei, mai scump per nucleu, dar prețul poate varia în funcție de performanță	Mai accesibil per nucleu, dar costul total poate fi mai mare în funcție de configurație și performanțe grafice
Execută instrucțiuni pentru procesare generală	Execută operații paralele pentru grafică și calcul intensiv

Este important să subliniez că CPU și GPU sunt proiectate pentru utilizări diferite, iar alegerea între ele depinde de tipul de sarcini care se doresc a fi realizate.

1.2. Programare paralelă pe plăci grafice. Aplicații

Datorită puterii mari de procesare paralelă, programarea pe GPU a găsit aplicații într-o gamă largă de domenii. Câteva dintre aceste domenii în care este folosită programarea pe GPU în cercetări recente sunt: inteligență artificială (AI – artificial intelligence) și învățare automată (ML – machine learning), analiza datelor de mari dimensiuni, simulări științifice, grafică și randare 3D, medicină și bioinformatică sau criptografie și securitate (**Figura 1.1**).

Figura 1.1 *Aplicații recente ale programării pe GPU* **(Baji 2018)**

GPU-urile permit transformarea și analiza rapidă a seturilor de date mari (big data) (Chen et al. 2018). Acest proces include analiza datelor în timp real, prelucrarea și filtrarea datelor și aplicarea de algoritmi de învățare automată pe seturi mari de date. Astfel, există cercetări în care se explorează

modul în care GPU-urile pot fi utilizate pentru accelerarea prelucrării datelor de mari dimensiuni. Ca exemplu, se analizează diferite tehnici de paralelizare și optimizare pentru a obține performanțe ridicate în analiza big data (Wu, Sun et al. 2021) (Kumar și Mohbey 2022). Se propun, de asemenea, algoritmi și tehnici de optimizare pentru a reduce timpul de execuție și a gestiona eficient memoria în operațiunile de analiză a datelor mari (Jiang et al. 2015).

Programarea pe GPU este utilizată în domeniul științelor computaționale pentru accelerarea calculului numeric intensiv (Prabhu et al. 2011). Acest lucru include simulări în fizică, chimie, biologie și alte domenii, unde se efectuează calcule complexe și iterative. În fizica computațională, de exemplu, accelerarea proceselor este de mare importanță pentru obținerea în timp real a rezultatelor dorite. Programarea GPU este o abordare potrivită pentru obținerea unui timp de execuție foarte bun atunci când este posibilă o paralelizare masivă (**Spridon**, Advances in CUDA for computational physics 2023). Astfel, deși mulți dintre algoritmii cunoscuți utilizați în fizica computațională au fost deja paralelizați și o parte dintre ei sunt adăugați la biblioteca CUDA (NVIDIA 2019), se caută încă noi metode de optimizare și de creștere a vitezei de execuție. Timpul de execuție este crucial în multe probleme complexe și, prin urmare, orice îmbunătățire în această direcție este încă necesară. Algoritmii hibrizi paraleli (CPU-GPU) sunt dezvoltați în mod continuu pentru a obține rezultate de calcul de înaltă performanță cu costuri minime (**Spridon**, Advances in CUDA for computational physics 2023).

1.3. Tehnologii pentru programare pe plăcile grafice – CUDA

Există mai multe tehnologii și platforme disponibile pentru programarea pe GPU. Dintre acestea cele mai importante sunt CUDA (Compute Unified Device Architecture), OpenCL (Open Computing Language), SYCL (Single-source Heterogeneous Programming in C++) , Vulkan.

În literatura de specialitate există o serie de lucrări în care se studiază comparativ tehnologiile de programare pe GPU. Astfel, Karimi et al. fac teste de performanță și compară timpii de transfer de date către și de la GPU, timpii de execuție a kernel-ului și timpii de execuție ai aplicațiilor end-to-end atât pentru CUDA, cât și pentru OpenCL pe o aceeași placă video (Karimi, Dickson și Hamze 2010). Rezultatele lor sunt prezentate în **Figura 1.2**. După cum se poate observa în respectivele teste CUDA a avut rezultate mai bune la transferul de date către și de la GPU. Nu s-a observat nicio schimbare considerabilă în performanța relativă a transferului de date pentru OpenCL, atunci când au fost transferate mai multe date. Execuția kernel-ului CUDA a fost, de asemenea, mai rapidă decât cea a OpenCL, chiar dacă cele două implementări au fost foarte similare.

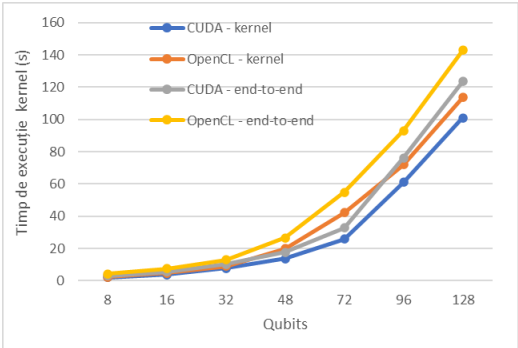


Figura 1.2 *Comparație timpi de execuție CUDA vs OpenCL*

Astfel, este demonstrat că arhitectura CUDA este o alegere mai bună pentru aplicațiile în care este nevoie de o performanță ridicată. În caz contrar, alegerea între CUDA celelalte tehnologii de programare pe GPU poate fi făcută luând în considerare factori precum familiaritatea anterioară cu oricare dintre sisteme, instrumentele de dezvoltare disponibile pentru hardware-ul GPU țintă sau portabilitatea aplicației rezultate. În această lucrare am ales arhitectura CUDA pentru performanța superioară anterior demonstrată în literatura de specialitate.

Un workflow al unui protocol de lucru CUDA este prezentat în **Figura 1.3**. Aplicațiile încep să ruleze pe CPU și codul *host* gestionează și codul de tip *device*. Datele care trebuie procesate sunt încărcate în memoria *host*, se alocă memoria necesară pe GPU și datele sunt încărcate în memoria acestuia folosind apeluri CUDA API, cum ar fi „cudaMalloc()" sau „cudaMemcpy()". Funcțiile kernel sunt apelate de pe CPU și rulează pe GPU, profitând de capacitatea GPU de a procesa sarcini intensive care pot fi executate în paralel. Pentru a lansa o funcție kernel, trebuie să specificăm numărul de fire de execuție și numărul de blocuri de utilizat. Acest lucru se face folosind sintaxa „<<<>>>" din CUDA. Odată ce kernel-ul a fost lansat, acesta se va executa pe GPU. Fiecare thread (fir de execuție) va executa același cod, dar cu date diferite. Datele pentru fiecare thread sunt accesate folosind indexul thread-ului, care este furnizat de CUDA. Pentru a vă asigura că toate firele de execuție și-au încheiat procesul de calcul înainte de a trece la pasul următor, firele de execuție trebuie sincronizate folosind funcția „__syncthreads()". După finalizarea execuției kernel-ului, datele trebuiesc transferate înapoi de la GPU pe CPU. În cele din urmă, memoria care a fost alocată pe GPU trebuie eliberată folosind „cudaFree()".

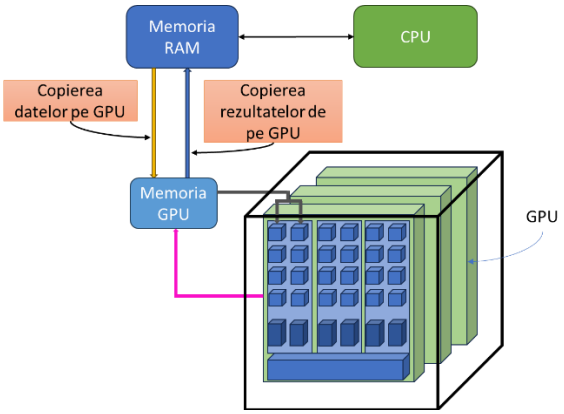


Figura 1.3 Etapele de lucru pentru un program CUDA

Managementul memoriei joacă un rol important pentru cele mai bune rezultate cu programarea CUDA. De asemenea, este necesar să se cunoască ierarhia memoriei GPU, astfel încât aceasta să poată fi folosită cât mai eficient posibil. Nivelurile de memorie ale GPU (memorie globală, memorie constantă, memorie partajată, memorie locală și registre) sunt prezentate în **Figura 1.4.**

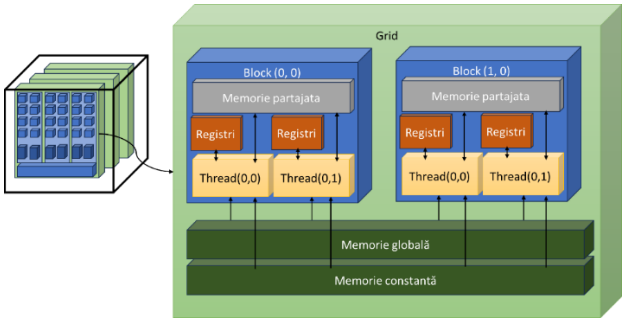


Figura 1.4 Ierarhia memoriei GPU

În domeniul calculului paralel și al aplicațiilor care necesită putere de calcul ridicată, CUDA a devenit o tehnologie populară. Cu ajutorul CUDA, programatorii pot exploata puterea de procesare masivă a GPU-urilor pentru a accelera rezolvarea problemelor complexe și pentru a obține performanță superioară într-o varietate de domenii.

Capitolul 2. Generare de rețele aleatoare

În acest capitol sunt prezentate două metode de generare de rețele aleatoare, necesare atunci când este vorba de studiul eficienței algoritmilor din teoria grafurilor. Metodele de generare propuse sunt paralelizate și sunt expuse rezultatele în ceea ce privește timpii de execuție și accelerarea în utilizării programării CUDA. Acest capitol se bazează pe lucrarea (Deaconu și **Spridon** 2021), la care sunt coautor.

2.1. Grafuri - generalități

Grafurile reprezintă o ramură importantă a matematicii, cât și a informaticii, care se ocupă cu studiul structurilor de relații între obiecte. Ele sunt utilizate pentru a modela și analiza interconexiunile între diverse entități sau elemente. Un graf este format dintr-un set de noduri sau vârfuri, reprezentate prin puncte, și arce sau muchii, care conectează noduri între ele. Teoria grafurilor se ocupă de studierea proprietăților, caracteristicilor și algoritmilor asociați grafurilor. Teoria grafurilor are o multitudine de aplicații în diferite domenii, inclusiv în informatică, rețele, optimizare, inteligență artificială, bioinformatică. Algoritmii de grafuri sunt utilizați pentru rezolvarea problemelor de căutare, traversare, conectivitate, planificare și multe altele. Prin studierea și aplicarea teoriei grafurilor, putem înțelege și analiza structurile complexe de relații între obiecte, găsi soluții eficiente la diverse probleme și dezvolta algoritmi optimizați pentru diferite scenarii.

Teoria grafurilor este un domeniu care investighează caracteristicile și comportamentul diverselor tipuri de grafuri, precum și dezvoltarea algoritmilor specializați în rezolvarea problemelor asociate grafurilor fiind o ramură a matematicii discrete.

Definiție 2.1 Un **graf** este o pereche ordonată, $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$, formată dintr-o mulțime, $\boldsymbol{V}$, de elemente numite noduri sau vârfuri și o mulțime de muchii (sau arce), $\boldsymbol{E}$, care conectează aceste noduri.

Definiția formală a unui graf poate varia în funcție de contextul în care este utilizat, dar în cele ce urmează prezint câteva elemente de bază din teoria grafurilor.

Într-un graf $G = (V, E)$ numărul de elemente din E sau cardinalul mulțimii V este numit ordinul lui G, iar numărul de elemente din E, sau cardinalul mulțimii E este numit dimensiunea lui G. Ordinul unui graf este de obicei notat cu n și dimensiunea lui G este notată cu m. Fiecare element din V este numit nod (sau vârf), iar fiecare element din E este numit muchie. Pentru o arc $a = (u, v)$, nodul u și nodul v sunt noduri adiacente; arcul a și nodul u (sau v) sunt incidente între ele. Pentru fiecare arc $a = (u, v)$, nodurile u și v sunt numite noduri terminale. O buclă este o arc $a = (u, v)$ ale cărei noduri terminale sunt identice, adică $u = v$. Arcele multiple sunt un set de muchii care au aceeași pereche de noduri terminale.

Definiție 2.2 Se numește ***graf aleatoriu*** un graf în care numărul de noduri, numărul de muchii și conexiunile dintre ele sunt generate în mod aleatoriu prin diferite metode.

Erdõs și Rényi au introdus grafurile binomiale aleatoare în lucrarea publicată în 1959 (Erdõs și Rényi 1959). Aceste grafuri aleatoare sunt generate pe baza valorilor a doi parametri: n (numărul de noduri) și $p \in [0, 1]$ - probabilitatea introducerii oricărei muchii în graf). Aceste tipuri de rețele aleatoare au fost aplicate pentru indicii Zagreb, indicele general de sumă-conectivitate, indicele general de sumă inversă și indicele general primul geometric-aritmetic (Cuadra și Nieto-Borge 2021). Într-o rețea generată în acest mod, există posibilitatea ca sursa să comunice prost cu nodul stoc sau chiar să nu comunice deloc. În 2002, Albert și Barabási au introdus modelul lor (BA) constând dintr-un algoritm bazat pe mecanismul de atașare preferențială pentru generarea de rețele aleatoare fără scală (Albert și Barabási 2002). Rețelele generate astfel au aplicație reală pe Internet, rețele de citare, World Wide Web și unele rețele sociale. Algoritmul începe cu o rețea având m_0 noduri date. Secvențial, nodurile sunt introduse în rețea. Fiecare dintre aceste noduri nou adăugate este conectat la $m \leq m_0$ noduri existente folosind o probabilitate dată, care este proporțională cu numărul de conexiuni pe care le aveau deja nodurile adăugate anterior. Probabilitatea p_i de a conecta un nou nod la nodul i este:

$$p_i = \frac{k_i}{\sum_j k_j} \tag{2.1}$$

Având în vedere faptul că rezultatele existente din literatura de specialitate despre rețele au de-a face cu grafuri specifice care nu sunt suficient de generale, sau neadecvate pentru problemele de flux în rețea, în lucrarea (Deaconu și **Spridon** 2021) am propus o nouă idee de generare a rețelelor aleatoare care prezintă avantajele că este rapidă și bazată pe proprietatea naturală a fluxului care poate fi descompus în drumuri și cicluri orientate elementare. În consecință, rețelele generate în acest mod sunt potrivite pentru testarea corectitudinii și eficienței algoritmilor pentru probleme de flux de rețea, cum ar fi fluxul de cost minim, fluxului maxim, problema fluxului multi-marfă etc.

2.2. CUDA în teoria grafurilor

Programarea pe GPU a arătat rezultate promițătoare în accelerarea algoritmilor din teoria grafurilor în ultimii ani. Câteva dintre cele mai recente cercetări în programarea GPU în teoria grafurilor sunt în următoarele direcții Rețele neuronale ale grafurilor (GNN) (Zonghan Wu, 2021) (Tianfeng Liu,

2023), partiționarea grafurilor (Santosh Nage, 2015), numărarea triunghiurilor dintr-un graf (Liu Hu, 2021), algoritmi pentru găsirea drumului cel mai scurt între două noduri dintr-un graf (Carl Yang, 2022).

În general, cele mai recente cercetări în programarea GPU pentru teoria grafurilor demonstrează potențialul GPU-urilor de a accelera algoritmii din teoria grafurilor și de a gestiona grafuri de dimensiuni foarte mari. Acest lucru poate duce la o performanță și o scalabilitate îmbunătățite pentru o gamă largă de aplicații, de la învățarea automată până la analiza rețelelor sociale sau probleme de rutare.

2.3. Algoritmi de generare a grafurilor aleatoare

Fie $G = (V, E, s, t, c, w)$ o rețea $s - t$, unde V este o mulțime care conține $n > 0$ vârfuri (noduri), iar E este o mulțime de $m \geq 0$ așa - numite arce (muchii orientate). Fiecare arc $a = (u, v) \in E$ leagă două noduri u și v din V, iar s este un nod special numit sursă și t este un nod numit nod stoc. În G, definim funcția de capacitate $c: E \to R_+^*$ și, respectiv, funcția de cost $w: E \to R_+$. Valoarea $c(a)$ este fluxul maxim care poate fi transportat de la nodul u la nodul v pe arcul $a = (u, v) \in E$, iar $w(a)$ este costul unitar al transportului fluxului pe arcul a.

Ahuja și coautorii prezintă următoarea teoremă (Ahuja, Magnanti și Orlin 1993):

Teorema 2.1 Orice flux admisibil poate fi descompus în drumuri și circuite astfel încât:

(a) Orice drum cu flux pozitiv conectează sursa s de nodul stoc t.

(b) Cel mult $n + m$ drumuri și circuite au flux diferit de 0. Dintre acestea cel mult m circuite au fluxuri nenule.

Demonstrația **Teoremei 2.1** se găsește în (Ahuja, Magnanti și Orlin 1993).

Comparațiile de corectitudine și eficiență ale algoritmilor pentru problemele de flux sunt importante atunci când sunt elaborate noi metode pentru rezolvarea acestora. Pentru a face acest lucru, este nevoie de un instrument rapid și de încredere care să genereze rețele aleatoare, începând cu cele simple și continuând cu rețele de mari dimensiuni. Am dezvoltat o metodă bazată pe modelul Erdõs–Rényi folosind ideea din Teorema 2.1 pentru a crea un astfel de instrument. Deoarece un flux poate fi descompus în fluxuri elementare, o abordare naturală este de a genera drumuri s-t și circuite elementare aleatoare.

În lucrarea (Deaconu și **Spridon** 2021) se prezintă algoritmi pentru generarea de drumuri și circuite elementare într-o rețea. Astfel, un prim algoritm pentru a genera un drum s-t elementar aleatoriu într-o rețea cu n noduri este **Algoritmul 2.1**.

Algoritmul 2.1 Algoritm de generare a unui drum s-t aleatoriu - v1 (AGDA1)

/ sursa este considerată ca fiind nodul cu index 0, iar nodul stoc este considerat nodul cu indexul n − 1 (ultimul nod)*/*
$s = 0$;
$t = n - 1$;
/ inițial doar nodul sursa aparține drumului */*
Pentru fiecare nod $j \neq s$ **execută**
    $pathnode[j] = false$;
sfârșit pentru;
$pathnode[s] = true$;
/ construirea drumului aleatoriu */*

```
u = s;
Pentru j= 1, n − 1 execută
        /* se alege aleatoriu un indice k, ca fiind indicele nodului următor care va fi adăugat la drum */
        k = random(0, n− j − 1);
        l = 0;
        /* se caută nodul v ca fiind al k-lea nod dintre nodurile neselectate încă*/
        Pentru v ∈ V execută
                Dacă pathnode[v] atunci
                        continuă;
                sfârșit dacă;
                Dacă l = k atunci
                        stop pentru;
                sfârșit dacă
                l = l + 1;
        sfârșit pentru;
        /* se adaugă arcul (u,v) la rețea */
        ma[u][v] = 1;
        /* se marchează nodul ca făcând parte din drum */
        pathnode[v] = true;
        /* dacă nodul ales este nodul stoc, atunci se oprește generarea*/
        dacă v = t atunci
                stop pentru;
        sfârșit dacă;
        /* nodul u devine nodul v pentru a pregăti adăugarea unui nou nod */
        u = v;
sfârșit pentru;
```

În AGDA1, fără a restrânge generalitatea algoritmului, considerăm că indicele sursei este egal cu 0, iar $n-1$ este indicele nodului stoc t. Algoritmul construiește un drum pornind de la s. Elementele vectorului pathnode sunt corespunzătoare fiecărui nod și fiecare dintre ele este setat true sau false dacă nodul corespunzător a fost sau nu adăugat în drumul creat. Astfel, inițial doar elementul nodului sursă este setat true. La fiecare iterație, un nod nou, care nu a fost adăugat anterior, este selectat aleatoriu și adăugat la sfârșitul drumului. Pentru aceasta, se alege aleatoriu un indice k, unde $0 \leq k < n-j-1$ (j este numărul de noduri deja adăugate la drumul curent) , ca fiind indicele nodului următor care va fi adăugat la drum. Se caută nodul v ca fiind al k-lea nod dintre nodurile neadăugate încă (ale căror valoare în pathnode este false) și se setează nodul respectiv ca făcând parte din drum. De fiecare dată când un nod nou, v, este adăugat la drum, arcul (u, v) este adăugat în rețea, adică valoarea matricei de adiacență m_a este setată la 1 pe poziția (u, v), unde u este penultimul nod adăugat. Algoritmul se termină când nodul stoc, t, este adăugat la drum.

Pentru generarea unui circuit aleatoriu (**Algoritmul 2.2**), algoritmul este prezentat mai jos.

Algoritmul 2.2 Algoritm de generare a unui circuit s-t aleatoriu - v1 (AGCA1)

```
/* se alege aleatoriu un nod inițial u0 */
u0 = random(0, n − 1);
Pentru fiecare nod j ≠ u0 execută
        cyclenode[j] = false;
sfârșit pentru;
/* construirea circuitului aleatoriu */
u = u0;
Pentru j = 0, n − 1 execută
        /* se alege aleatoriu un indice k, ca fiind indicele nodului următor care va fi adăugat la circuit */
        k = random(0, n − j − 1);
```

```
        l = 0;
        /* se caută nodul v ca fiind al k-lea nod dintre nodurile neselectate încă*/
        Pentru fiecare nod v execută
                dacă cyclenode[v] atunci
                        continuă;
                sfârșit dacă;
                Dacă l = k atunci
                        stop pentru;
                sfârșit dacă;
                l = l + 1;
        sfârșit pentru;
        /* se adaugă acul (u, v) la rețea */
        ma[u][v] = 1;
        /* se marchează nodul ca făcând parte din circuit */
        cyclenode[v] = true;
        /* dacă nodul ales este același cu nodul de inițial atunci se oprește generarea*/
        Dacă v = u0 atunci
                stop pentru;
        sfârșit dacă;
        /* nodul u devine nodul v pentru a pregăti adăugarea unui nou nod */
        u = v;
sfârșit pentru;
```

În AGCA1, un circuit este construit începând cu un nod $u0$, ales aleatoriu. Vectorul $cyclenode$ are n elemente și sunt setate cu true sau false dacă nodul corespunzător a fost sau nu adăugat în circuitul curent. La fiecare iterație, un nou nod care nu face deja parte din circuit este selectat aleatoriu și adăugat la circuit. Acest lucru se realizează generând un număr aleatoriu k, $0 \leq k < n - j$. Ulterior se caută v, cel de-al k-ulea nod neadăugat deja în circuit și marcăm $cyclenode[v] = true$. De asemenea, de fiecare dată când un nou nod v este introdus în circuit, arcul (u, v) este, de asemenea, adăugat la rețea, unde u este nodul adăugat anterior circuitului. Algoritmul se termină când nodul $u0$ este adăugat din nou la circuit.

Algoritmii AGDA1 și AGCA1 pot construi în mod natural drumuri $s - t$ și circuite elementare. Complexitatea lor este $O(n^2)$. Acești doi algoritmi ar putea fi folosiți împreună pentru a construi rețele aleatoare. Cu toate acestea, vom prezenta mai jos o abordare mai rapidă.

Richard Durstenfeld propune un algoritm pentru a genera aleatoriu o permutare (Durstenfeld 1964). În **Algoritmul 2.3**, se propune o abordare similară, dar mai simplă, pentru a genera un vector amestecat de noduri având indicii între i_{start} și i_{end} (Deaconu and Spridon 2021).

Algoritmul 2.3 Algoritm pentru amestecarea vectorului de noduri (AAVN)

```
Parametri de intrare: i_start, i_end;

/* vectorul de noduri conține inițial indecșii de la i_start până la i_end*/
Pentru j = i_start, i_end execută
        nodes[j] = j;
sfârșit pentru;
/* se amestecă vectorul de noduri */
Pentru k = i_start, i_end execută
        u = random(i_start, i_end);
        v = random(i_start, i_end);
        Dacă u ≠ v atunci
```

$swap = nodes[u];$
$nodes[u] = nodes[v];$
$nodes[v] = swap;$
sfârșit dacă;
sfârșit pentru;

Având în vedere cele menționate anterior, introduc acum două metode noi pentru a genera drumuri s-t și circuite aleatoare elementare folosind AAVN.

Algoritmul 2.4 Algoritm de generare a unui drum s-t aleatoriu – v2 (AGDA2)

/ generare eficientă a unui vector de noduri fără s și t */*
$AAVN(1, n-2);$
$s = 0;$
$t = n - 1;$
/ generarea aleatorie a lungimii drumului */*
$l_{path} = random(2, n);$
/ se adaugă primul arc care se formează între nodul sursa și primul nod din vectorul de noduri amestecate ale rețelei */*
$ma[s][nodes[1]] = 1;$
Pentru $k = 1, l_{path} - 3$ **execută**
$ma[nodes[k]][nodes[k+1]] = 1;$
sfârșit pentru;
$\boldsymbol{ma[nodes[l_{path} - 2]][t] = 1;}$

În **Algoritmul 2.4** (AGDA2) mai întâi se generează o permutare a vectorului de noduri, din care s-au scos nodurile sursă și stoc. Aleatoriu se generează lungimea drumului $0 \leq l_{path} \leq n$. Ulterior, se creează drumul pornind cu arcul dintre nodul sursă și primul nod din vectorul de noduri amestecate și, treptat, se adaugă arcele create din perechile de noduri consecutive ale vectorului, până pe poziția $l_{path} - 2$. Ultimul arc adăugat în drum este cel dintre nodul de pe poziția $l_{path} - 2$ din vector și nodul stoc.

Algoritmul 2.5 Algoritm de generare a unui circuit aleatoriu – v2 (AGCA2)

/ generare eficientă a unui vector de noduri */*
$\boldsymbol{AAVN(0, n-1);}$
/ generarea aleatorie a lungimii circuitului */*
$\boldsymbol{l_{cycle} = random(2, n);}$
/ se adaugă arcele care se formează între primele* $\boldsymbol{l_{cycle}}$ *noduri din vectorul de noduri amestecate ale rețelei */*
Pentru $\boldsymbol{k = 0, l_{cycle} - 2}$ **execută**
$ma[nodes[k]][nodes[k+1]] = 1;$
sfârșit pentru;
$ma[nodes[l_{cycle} - 1]][nodes[0]] = 1;$

În **Algoritmul 2.5**, AGCA2, se generează inițial o permutare a vectorului de noduri. Lungimea circuitului, l_{cycle} , se generează aleatoriu cu valori între 2 și n. Circuitul este creat adăugând arcele formate din primele $l_{cycle} - 1$ noduri consecutive din vectorul amestecat. Ultimul arc adăugat la circuit este între cel de-al $l_{cycle} - 1$ -lea nod din vector și primul nod din vector.

Mai jos este prezentat **Algoritmul 2.6** de generare a unei rețele de flux aleatorie.

Algoritmul 2.6 *Algoritm de generare a unei rețele s-t de flux, aleatorie (AGRFA)*

Parametri de intrare: $p, n_{path}, n_{cycle}, min_c, max_c, min_w, max_w$;

/ generarea a "n_{path}" drumuri aleatorii */*
Pentru k = 1, n_{path} **execută**
    $AGDA2$;
sfârșit pentru ;
/ generarea a "n_{cycle}" circuite aleatoare */*
Pentru k = 1, n_{path} **execută**
    ARCA2;
sfârșit pentru;
/ generarea listei de adiacență la folosind matricea de adiacență ma */*
Pentru $i = 0, n-1$ **execută**
    $la[i] = null$;
sfârșit pentru;
/ atribuirea de arce de capacitați și costuri aleatoare arcelor, atunci când acestea sunt adăugate în "la" */*
Pentru $i = 0, n-1$ **execută**
    Pentru $j = 0, n-1$ **execută**
/ generarea arcelor conform modelului Erdős–Rényi */*
        Dacă $ma[i][j] = 0$ **și** $random(0, 1000) < p * 1000$ **atunci**
            $ma[i][j] = 1$;
        sfârșit dacă;
        Dacă $ma[i][j] = 1$ **atunci**
            adaugă $(j, random(min_c, max_c), random(min_w, max_w))$ în $la[i]$;
        sfârșit dacă;
    sfârșit pentru;
sfârșit pentru;

Teorema 2.3 Complexitatea AGRFA este $O(n \cdot max\{n_{path}, n_{cycle}, n\})$.

De obicei, este suficient să se ia în considerare numărul de drumuri și numărul de circuite mai mici decât numărul de noduri. Deci, în practică, cel mai probabil complexitatea este $O(n^2)$.

Complexitatea din **Teorema 2.3** poate fi îmbunătățită dacă generarea drumurilor, a circuitelor și a listelor de adiacență sunt paralelizate. Calculele din algoritm sunt elementare și implică doar valori întregi. Deci, AGRFA poate fi paralelizat în mod natural pe GPU-uri. Deoarece viteza de generare a rețelelor aleatoare de mari dimensiuni este esențială, îmbunătățirea complexității prin paralelizare poate juca un rol important. Luând în considerare un total de g GPU-uri, generarea drumurilor și a circuitelor poate fi împărțită în $max\{1, (n_{path} + n_{cycle}) / g\}$ grupuri. Generarea listelor de adiacență poate fi, de asemenea, împărțită în $max\{1, n/g\}$ grupuri. Deci, complexitatea implementării paralele pe GPU-uri a AGRFA este $O(n \cdot max\{n_{path}, n_{cycle}, n\} / g)$, dar la care se adaugă timpii de comunicare și acces a memoriei de pe GPU.

2.4. Rezultate și discuții

În **Figura 2.1.** au fost generate și afișate trei rețele având 6, 20 și, respectiv, 100 de noduri. Pentru prima rețea au fost generate 3 drumuri și 2 circuite. Pentru cea de-a doua rețea au fost generate 10 drumuri și 2 circuite, iar pentru ultima rețea, au fost generate 20 de drumuri și 10 circuite.

Au fost efectuate diferite teste pentru a ilustra creșterea timpului de generare odată cu scalarea rețelelor aleatoare, având numărul de noduri cuprins între 10 și 10 000. Numărul de noduri împreună cu numărul de drumuri și circuite considerate influențează direct viteza de generare a rețelei.

A fost folosit un Asus ROG Strix G17 G712LV, procesor Intel Core i7-10750H până la 5,10 GHz, 16 GB RAM, NVIDIA GeForce RTX 2060 6 GB GDDR6 cu 1920 de nuclee CUDA.

Paralelizarea a fost implementată folosind programarea CUDA pe GPU. Fiecare drum și fiecare circuit a fost creat pe un fir de execuție diferit. În plus, crearea listelor de adiacență din matricea de adiacență a fost paralelizată, lista pentru fiecare nod fiind obținută pe un fir diferit. Testele au arătat că utilizarea paralelizării devine din ce în ce mai eficientă odată cu creșterea dimensiunii rețelelor. Pentru rețelele mici (mai puțin de 50 de noduri) este mai bine de folosit implementarea algoritmului pe CPU, dar când numărul de noduri ale rețelelor este mai mare de 50, se preferă implementarea CUDA, rezultând o accelerare clară, de până la 19 ori mai rapid decât implementarea CPU. Speed-up-ul este o măsură a performanței relative a implementării paralele față de cea secvențială. Acesta este definit ca raportul dintre timpul de execuție pentru implementarea în mod secvențial și timpul de execuție în cazul implementării paralele.

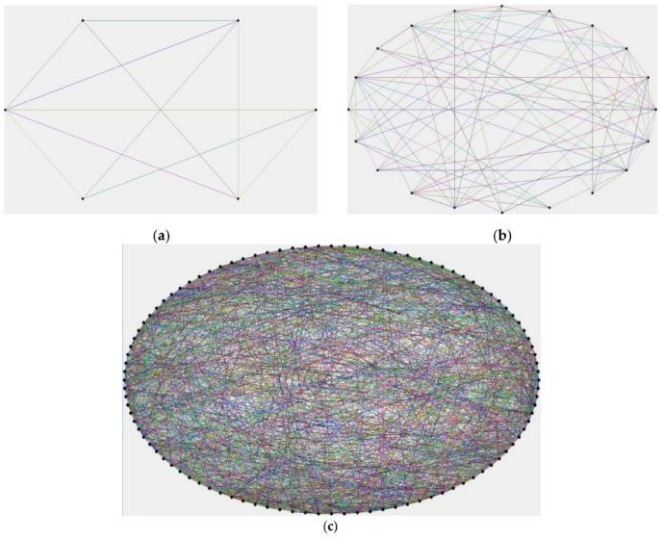


Figura 2.1 *Rețele generate folosind AGFFA. (a) Rețea cu n = 6, n_{path} = 3, n_{cycle} = 2; (b) Rețea cu n = 20, n_{path} = 10, n_{cycle} = 2; (c) Rețea cu n = 100, n_{path} = 20, n_{cycle} = 10.* (Deaconu și **Spridon** 2021)

În **Figura 2.2** este prezentată evoluția speed-up-ului pentru generarea de rețele aleatoare de diferite dimensiuni. După cum se poate observa, pentru rețele de dimensiuni mici, rularea pe GPU duce la o scădere a vitezei de execuție, cel mai probabil datorată timpilor de comunicare între CPU și GPU. Pe măsură ce dimensiunea rețelei crește, crește și factorul de accelerare datorat paralelizării masive pe GPU până la obținerea unui timp de execuție de 19 ori mai mic în cazul unei rețele cu 10000 de noduri atunci când se rulează folosind CUDA.

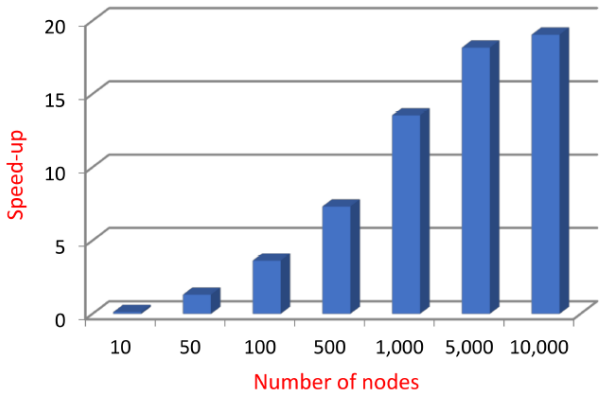


Figura 2.2 *Speed-up CPU/ CUDA* **(Deaconu și Spridon 2021)**

Analiza evoluției speed-up-ului pentru generarea de rețele aleatoare de diferite dimensiuni atunci când se folosește CUDA arată cum performanța sistemului variază în funcție de dimensiunea problemei. După cum era de așteptat cu cât dimensiunea problemei crește, cu atât avantajele paralelizării cu CUDA devin mai evidente. Aceasta se datorează capacității GPU-ului de a procesa în paralel o cantitate mare de date, ceea ce poate duce la îmbunătățiri semnificative ale timpului de execuție în comparație cu implementările secvențiale pe CPU.

Capitolul 3. Căutarea drumului cu pierderi minime într-o rețea de mari dimensiuni

În acest capitol, introduc și rezolv o problemă practică numită problema drumului cu pierdere minimă sau drumul cu rata de livrare maximă. Această problemă constă în găsirea drumului de la un nod sursă, s, la un alt nod dat, t, numit stoc (sink) într-o rețea generalizată, care are un factor câștig / pierdere atașat fiecărui arc, astfel încât pierderea să fie minimă între toate drumurile $s - t$. Acesta se bazează pe lucrarea (Deaconu, Spridon și Ciupala 2023), la care sunt coautor.

3.1. Context științific

Problema clasică pentru flux maxim presupune găsirea fluxului maxim care poate fi transportat de la un nod sursă la un așa-numit nod stoc într-o rețea în care fiecare arc are o capacitate $c(u, v), (u, v)\ \epsilon\ E$. De exemplu, o companie de furnizare a gazelor naturale poate dori să maximizeze cantitatea de gaz natural trimisă între o două de orașe prin rețeaua sa de conducte. Fiecare conductă din rețea are în mod evident o capacitate limitată.

În problema generalizată, fiecare arc, pe lângă capacitatea corespunzătoare, poate să aibă și un factor de pierdere sau de câștig care trebuie luat în considerare atunci când se calculează fluxul maxim. Altfel spus, problema generalizată pentru determinarea fluxului maxim este o extensie a problemei clasice a fluxului maxim într-o rețea în care, pentru a determina cantitatea maximă de flux, trebuie să se ia în considerare și alți factori, cum ar fi costurile sau capacitățile variabile ale arcelor.

Problema inversă generalizată a fluxului maxim (IGMF – Inverse Generalized Maximum Flow Problem) a fost introdusă și studiată în (Tayyebi și Deaconu 2019). În aceasta problemă se încearcă modificarea capacităților arcelor astfel încât un anumit flux admisibil să devină flux maxim în rețeaua modificată, iar distanța dintre vectorul inițial de capacități și vectorul modificat de capacități să fie minimă.

Din câte am studiat, aceste două probleme sunt singurele probleme de optimizare care au fost analizate în rețele cu pierderi sau câștiguri pe arce.

3.2. Problema găsirii drumului cu pierdere minimă

Fie $\mathrm{G} = (\mathrm{V}, \mathrm{E})$ un graf orientat, unde V este o mulțime finită în care elemente sunt numite vârfuri sau noduri și E este o mulțime de perechi ordonate de vârfuri, numite arce sau muchii orientate ($E \subseteq V \times V$). Vom considera în continuare că graful G este o reprezentare matematică a unei rețele de transport din viața reală (de apă, canalizare, gaz, electricitate etc.), unde arcele reprezintă liniile de transport, iar nodurile sunt modul în care se intersectează liniile de transport unele cu altele. În viața reală, atunci când se utilizează rețele de transport, în mod obișnuit, există pierderi pe arce din varii motive cum ar fi: din cauza evaporării, scurgerilor, disipării de energie, furtului etc. Pentru a modela matematic acest lucru, luăm în considerare pentru fiecare arc $(u, v) \in E$ un coeficient de pierdere, sau o rată de livrare notată cu $\alpha(u, v) \in (0, 1]$. Astfel, dacă x unități intră din nodul u pe arcul (u, v) , atunci în nodul v vor ajunge $x \cdot \alpha(u, v) < x$ unități.

În cele ce urmează, voi prezenta o metodă de calculare a drumului cu pierdere minimă (minimim loss path - MLP) sau drumului cu rata de livrare maximă (maximum delivery rate path - MDP) de la un nodul sursă, s, la nodul stoc, t. Notăm această problemă ca MLPP (minimul loss path problem). Pentru a rezolva MLPP, trebuie să identificăm un drum în G de la s la t, astfel încât rata de livrare de la s la t sa fie maximă dintre toate drumurile din Π, adică trebuie să găsim soluția următoarei probleme de optimizare:

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

Pentru rezolvarea problemei de mai sus se procedează după cum urmează: se transforma problema (3.2) într-o problema de minimizare astfel:

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

unde baza pentru logaritm este mai mare de 1, de exemplu, baza poate fi e sau 10.

Problema anterioară poate fi rescrisă sub forma:

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

unde:

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

Având în vedere faptul că valorile β atașate arcurilor sunt pozitive, este ușor de observat acum că problema (3.2) a fost redusă la o problemă clasică de optimizare pentru găsirea drumului cel mai scurt în rețeaua $G = (V, E, \beta)$. Această problemă poate fi rezolvată eficient folosind algoritmul lui Dijkstra într-o complexitate de timp de $O(n^2)$ sau $O(m \cdot log(n))$, în funcție de implementare (Schrijver 2012) (Fredman și Tarjan 1984), unde n denotă numărul de noduri ($n = |V|$), iar m reprezintă numărul de arce ($m = |E|$). În consecință, **Algoritmul 3.1** prezentat mai jos calculează MLP în G.

Algoritmul 3.1 Algoritmul de determinare a drumului cu pierdere minimă într-o rețea

Parametri de intrare:

- un graf orientat $G = (V, E)$
- $\alpha(u, v) \in (0,1], (u, v) \in E$

Parametri de ieșire:

- MLP în G

/ calcularea funcției $\beta(i, j)$ */*

Pentru $k = 0, m - 1$ **execută**

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

Sfârșit pentru

Dacă t nu este accesibil din s **atunci**

MLPP nu are soluție

Altfel

se aplică algoritmul pentru determinarea drumului cel mai scurt din s în $G = (V, E, \beta)$

fie $P = (s = v_0, v_1, \dots, v_k = t)$ cel mai scurt drum din s în t, atunci P este MLP în $G = (V, E)$ de la s la t

Sfârșit dacă.

Vom investiga, în cele ce urmează, cazul mai general, când, pe unele arce, ar putea exista câștiguri în loc de pierderi. Aceste câștiguri pot fi obținute, de exemplu, prin injectarea în rețea pe anumite arce (o nouă sursă de gaz, un prosumator în rețeaua electrică, etc). Astfel, în loc să setăm $\alpha(u, v) > (0, 1]$, am putea considera $\alpha(u, v)$ ca având valori pozitive, iar $\alpha(u, v) > 1$ pe un arc (u, v) dacă și numai dacă există un câștig pe acest arc.

Această problemă de optimizare are același model matematic (3.1) și poate fi, de asemenea, transformată în problema de minimizare (3.3). Totuși, se observă că valorile lui $\beta(a_k) = -\log(\alpha(a_k))$, pot fi negative (pe arcele pentru care $\alpha(u, v) > 1$). Mai mult, dacă în rețeaua astfel obținută există un circuit negativ, acesta corespunde unui circuit cu câștig infinit (produsul valorilor α pe un astfel de circuit este mai mare decât 1). Pe un drum $s - t$ care conține un astfel de circuit nu se poate găsi o rată maximă de livrare, deoarece pe respectivul drum fluxul poate fi mărit infinit prin trecerea de infinit ori pe acel circuit.

Algoritmul 3.2 rezolvă MLPP în cazul generalizat (în rețele în care pot exista atât câștiguri cât și pierderi pe arce).

Algoritmul 3.2 Algoritmul de determinare a drumului cu pierdere minimă într-o rețea generalizată

Parametri de intrare:

- un graf orientat $G = (V, E)$
- $\alpha(i, j) \in (0, \infty], (i, j) \in E$

Parametri de ieșire:

- MLP în G

/* *calculareaa funcției* $\beta(i,j)$ */
Pentru $k = 0, m - 1$ **execută**
$\beta(a_k) = -\log(\alpha(a_k)),\ a_k \in E$
Sfârșit pentru
Dacă t nu este accesibil din s **atunci**
MLPP nu are soluție
Altfel
se aplică algoritmul Bellman-Ford din s în $G = (V, E, \beta)$
Dacă G are circuit negativ **atunci**
MLPP nu are soluție
Altfel
fie $P = (s = v_0, v_1, \dots, v_k = t)$ cel mai scurt drum din s în t, atunci P este MLP în $G = (V, E)$ de la s la t
Sfârșit dacă
Sfârșit dacă

Ca și în cazul algoritmului anterior, în **Algoritmul 3.2** se calculează inițial funcția β pentru fiecare arc al rețelei. Se realizează ulterior testul de fezabilitate pentru MLPP. În cazul în care nodul stoc, t, este accesibil din nodul sursă, s, se aplică algoritmul Bellman-Ford pentru determinarea celui mai scurt drum în rețeaua nou formată $G = (V, E, \beta)$. În cazul în care în rețea se identifică un circuit negativ, atunci problema găsirii drumului cu pierdere minimă nu are soluție, altfel, drumul găsit este și drumul cu pierderi minime în rețeaua inițială.

3.3. Algoritmi pentru determinarea celui mai scurt drum într-o rețea

Algoritmii pentru determinarea drumului minim sunt metode utilizate în teoria grafurilor și în calculul operațional pentru a găsi cel mai scurt drum între două puncte (noduri) într-un graf. Acești algoritmi sunt esențiali în diverse domenii, cum ar fi rețelele de comunicații, transport, logistică și inteligența artificială.

Algoritmul lui Dijkstra este un algoritm pentru găsirea celui mai scurt drum într-un graf care are doar costuri pozitive ale arcelor (Dijkstra 1959). Algoritmul lui Dijkstra este utilizat pe scară largă în informatică și inginerie pentru a găsi drumuri optime în rețelele de transport, rutarea internetului și multe alte aplicații. Algoritmul lui Dijkstra nu poate fi aplicat pentru rețele cu valori negative ale arcelor.

Algoritmul **Bellman-Ford** se poate aplica pentru a determina drumul cel mai scurt într-un graf care conține arce cu valori negative ale costurilor (Bellman 1958). În plus, acest algoritm poate decide și dacă rețeaua conține sau nu circuite negative (cu câștiguri infinite). În acest caz, problema nu este fezabilă.

Având în vedere dimensiunea rețelelor în aplicațiile din viața reală, timpul de execuție pentru algoritmii propuși este foarte important. Astfel, a fost abordată o metoda de paralelizare folosind programarea pe GPU prin intermediul CUDA (Compute Unified Device Architecture) pentru algoritmii propuși. Algoritmii Dijkstra și Bellman-Ford au fost implementați anterior pe arhitectura CUDA (Harish și Narayanan 2007, Ortega-Arranz, et al. 2013, Surve și Shah 2017). Au fost utilizate diverse tehnici de paralelizare și au fost obținute creșteri semnificative de viteză în comparație cu implementările CPU. Am adaptat aceste abordări pentru calcularea MLP-urilor.

În algoritmul lui Dijkstra, în fiecare iterație i, calculează distanța minimă dintre sursă și nodurile care aparțin mulțimii de *noduri nesetate* (noduri pentru care distanța minimă nu a fost încă determinată), Ui. Unul dintre nodurile pentru care distanța este egală cu această valoare minimă este

setat și devine nodul de analizat. Arcele de ieșire ale nodului de analizat sunt parcurse pentru a relaxa distanțele corespunzătoare nodurilor adiacente. Pentru a paraleliza algoritmul Dijkstra, este necesar să se identifice care noduri pot fi utilizate ca noduri de analizat simultan. Există o serie de lucrări în care mulțimea nodurilor de analizat a fost stabilită în diferite moduri. De exemplu, în lucrarea (Martin, Torres și Gavilanes 2009) în mulțimea nodurilor de analizat se propune inserarea tuturor nodurilor care au distanța egală cu distanță minimă. Ortega-Arranzet et al propun o îmbunătățire, adăugând în mulțimea nodurilor de analizat și noduri care au distanța mai mare decât distanța minimă determinată (Ortega-Arranz, și alții 2013). Algoritmul calculează în fiecare iterație i, pentru fiecare nod al mulțimii de noduri nesetate, $u \in Ui$, suma dintre distanța calculată până atunci și costurile arcelor sale incidente către exterior. Ulterior, se determină minimul dintre aceste valori calculate. Într-un final, acele noduri a căror distanță este mai mică sau egală decât această valoare minimă determinată la pasul anterior sunt setate și inserate în mulțimea nodurilor de analizat. Se definește Δi ca fiind valoarea minimă calculată în fiecare iterație i și care susține că orice nod nesetat u care are distanța $\delta(u) \leq \Delta i$ poate fi setat. Cu cât valoarea Δi este mai mare, cu atât este mai exploatat paralelismul. Cu toate acestea, în funcție de graful procesat, utilizarea unui Δi foarte optimist poate duce către calcule care distrug orice câștig de performanță față de execuția secvențială.

Algoritmul 3.3 reprezintă pseudocodul algoritmului Bellman – Ford paralelizat pentru implementarea pe GPU. Astfel, prima etapă este cea de inițializare care are loc pe GPU. Urmează etapa de relaxare în care se verifică dacă există vreun alt drum mai scurt de la nodul sursă la un nod dat. Pentru această etapă se apelează funcția kernel - **Algoritmul 3.4.**

Algoritmul 3.3 Pseudocodul pentru algoritmul Bellman – Ford paralelizat pe GPU

Parametri de intrare: un graf orientat $G = (V, E)$
Parametri de ieșire: MLP în G

<<<initializare>>>(dist)
steps = 0
Repetă
$\text{dist}_{aux} = dist, a_k \in E$
$<<< Bellman - Ford\ kernel >>> (G, dist, dist\_aux)$
$steps = steps + 1$
până când $dist\_aux = dist$ **sau** $steps = n - 1$

Fiecare kernel (**Algoritmul 3.4**) execută câte un thread GPU pentru fiecare nod v cu indexul id, calculând distanța minimă. Pentru aceasta se folosesc cele mai scurte drumuri calculate anterior pentru predecesorii nodurilor. Dacă se găsește un nou drum, mai scurt pentru v, distanța este actualizată pentru nodul v. Astfel la fiecare iterație se calculează un nou vector de distanțe, care la sfârșitul iterației înlocuiește vechiul vector de distanțe. Algoritmul se oprește atunci când cei doi vectori ai distanțelor sunt la fel sau se găsește un circuit de cost negativ.

Algoritmul 3.4 Kernel-ul Bellman-Ford

Parametri de intrare:

- un graf orientat $G = (V, E)$
- dist – vectorul de distante
- dist_aux –vectorul de distanțe auxiliar

$tid = threadId$
//caută distanța cea mai scurtă de la nodul sursă la nodul tid
$min = INF$
Pentru toți predecesorii i ai nodului tid **execută**
    Dacă $w[i, tid] + dist\_aux[i] < min$ **atunci**
        $min = w[i, tid] + dist\_aux[i]$
    sfârșit dacă
sfârșit pentru
Dacă $min < dist\_aux[tid]$
    $dist[tid] = min$
sfârșit dacă

3.4. Rezultate și discuții

Pentru testarea algoritmilor pentru MLPP, au fost generate rețele aleatoare folosind metoda din lucrarea (Deaconu și **Spridon** 2021). Testele au fost efectuate pe un sistem Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz 2.59, având GPU NVIDIA GeForce RTX 2060 și OS Windows 10. Rețelele aleatoare analizate au un număr de noduri între 2000 și 50000 și un număr variat de arce după cum au fost generate cu ajutorul **Algoritmul 2.6**. Timpii de execuție pentru rețelele de dimensiuni reduse sunt foarte mici, iar utilizarea programării GPU nu este necesară. Prin creșterea numărului de noduri, viteza de crește execuție până la 390 pentru rețelele cu 40000 de noduri și până la 326M de arce, așa cum se poate observa în **Figura 3.**.

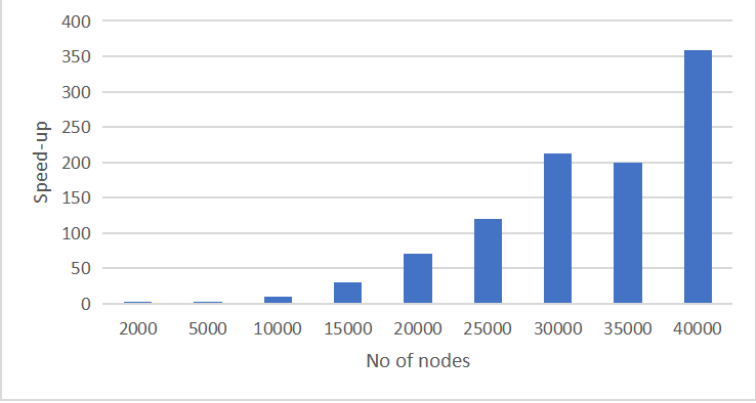


Figura 3.1 *Creșterea vitezei de execuție (speed-up-ului) pe GPU pentru* **Algoritmul 3.1** *pe rețele dense cu număr variat de noduri* (Deaconu, **Spridon** și Ciupala 2023)

În cazul implementării **Algoritmul 3.2** la baza căruia stă algoritmul Bellman-Ford timpul de execuție crește atunci când crește numărul de noduri și densitatea arcului de rețea, cea mai mare viteză pe GPU este de 5,8 și este înregistrată pentru o rețea cu 10000 de noduri și 24M de arce (**Figura 3.**).

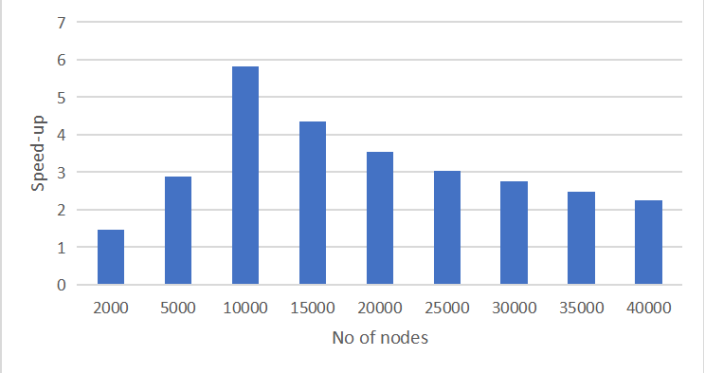


Figura 3.2 *Creșterea vitezei de execuție (speed-up) pe GPU pentru* ***Algoritmul 3.2*** *pe rețele dense cu număr variat de noduri* **(Deaconu, Spridon și Ciupala 2023)**

În general, timpul de execuție pe GPU este semnificativ mai mic decât cel pe CPU pentru toate dimensiunile rețelelor. Acest lucru subliniază beneficiile aduse de paralelizarea cu CUDA. Chiar dacă timpul de execuție pe GPU crește odată cu dimensiunea rețelei, acest lucru se întâmplă într-un ritm mai lent decât timpul de execuție pe CPU (**Figura 3.**).

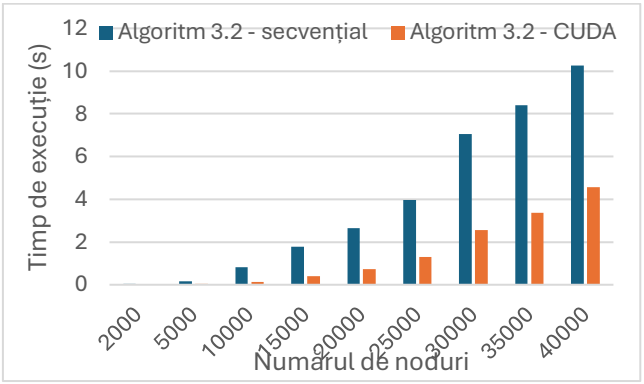


Figura 3.3 *Timpul de execuție pentru Algoritmul 3.2 pentru rețele dense*

De asemenea, odată cu creșterea densității rețelei crește și timpul de execuție pentru implementarea secvențială, în timp ce, pentru implementarea paralelă creșterea timpului de execuție mai lentă odată cu creșterea densității rețelei (**Figura 3.4**).

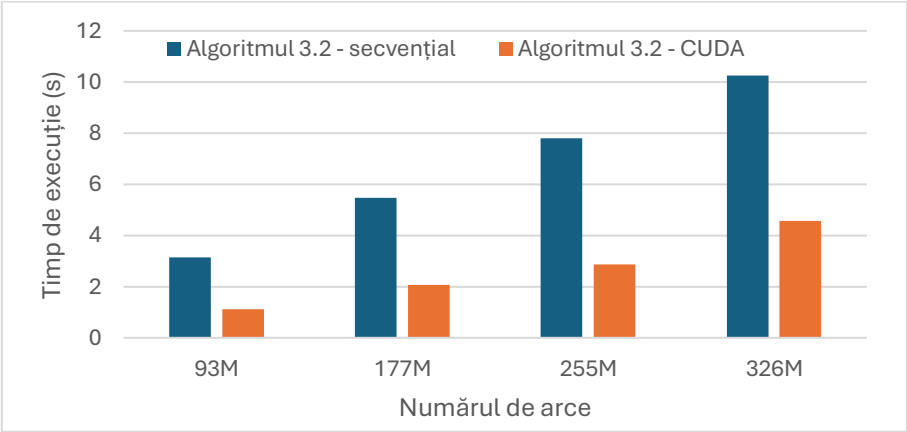


Figura 3.4 *Evoluția timpului de execuție pentru* ***Algoritmul 3.2*** *în funcție de densitatea rețelei*

Valorile speed-up-ului sunt semnificative, indicând beneficiile paralelizării cu CUDA pe măsură ce rețelele devin mai mari.

Interpretarea generală a rezultatelor arată că paralelizarea cu CUDA aduce îmbunătățiri semnificative de performanță pentru algoritmul Bellman-Ford, mai ales pentru rețelele de dimensiuni mari.

În concluzie, rezultatele arată o viteză de execuție crescută atunci când se utilizează programarea GPU pentru **Algoritmul 3.1** pentru rețele mari și dense. Deși nu la fel de semnificativă, o îmbunătățire a timpului de execuție a fost obținută și pentru **Algoritmul 3.2**, folosind algoritmul Bellman-Ford în implementarea bazată pe GPU.

Comparând rezultatele corespunzătoare celor doi algoritmi, putem observa diferențe și similarități în performanța acestor. Astfel, în general, ambii algoritmi prezintă un speed-up semnificativ în cazul paralelizării cu CUDA. Ambii algoritmi beneficiază de timpi de execuție mai mici pe GPU în comparație cu CPU pentru rețele de diferite dimensiuni. Cu toate acestea, se pot observa diferențe în modul în care timpul de execuție variază odată cu creșterea dimensiunii rețelei pentru fiecare algoritm. De exemplu, timpul de execuție pentru **Algoritmul 3.1** pe GPU poate crește mai rapid odată cu dimensiunea rețelei, în timp ce algoritmul Bellman-Ford pare să aibă o creștere mai lentă a acestuia (**Figura 3.**). Pentru ambii algoritmi beneficiile paralelizării pe GPU devin mai evidente pe măsură ce dimensiunea problemei crește. Performanța relativă a celor doi algoritmi poate varia în funcție de caracteristicile rețelelor și de alte considerente specifice aplicației. Algoritmul Dijkstra este cunoscut pentru complexitatea sa inferioară în comparație cu Bellman-Ford, ceea ce poate influența performanța relativă în funcție de anumite caracteristici ale rețelelor.

Capitolul 4. Determinarea fluxului cu pierderi minime într-o rețea generalizată

În acest capitol prezint o posibilă aplicație a algoritmilor de găsire a drumului cu pierdere minimă pentru determinarea fluxului maxim într-o rețea generalizată (cu pierderi / câștiguri pe arce). În acest sens algoritmul Ford-Fulkerson a fost adaptat în așa fel încât, succesiv, se determină drumuri s-t cu pierdere minimă. Prezint două posibile implementări ale algoritmului: secvențial și, respectiv, folosind paralelizare pe GPU. Pentru cele două implementări au fost rulate multiple teste, făcându-se o comparație a timpilor de execuție. Aceste rezultate au fost acceptate pentru prezentare la conferința 21st International Conference on Applied Computing (AC 2024) (**Spridon**, Deaconu and Popa, et al. 2024).

4.1. Problema tradițională a fluxului maxim

În problema fluxului maxim, obiectivul este de a trimite cât mai mult flux posibil între două noduri, respectând limitele de capacitate ale arcelor. O instanță a problemei fluxului maxim este o rețea $G = (V, E, s, t, c)$ antisimetrică, unde $s \in V$ este nodul sursă, $t \in V$ este un nodul stoc, iar c este o funcție capacitate.

Teorema 4.1 Un flux este maxim dacă și numai dacă în rețeaua reziduală nu există drumuri de mărire a fluxului.

Demonstrația Teoremei 4.1 se găsește în lucrarea (Ford și Fulkerson 1956).

O *rețea reziduală* este o rețea $G_f = (V, E, c_f)$, unde $c_f: E \to \Re$, $c_f(u,v) = c(u,v) - f(u,v)$, este funcția capacitate reziduală. De exemplu, dacă $c(u,v) = 40$, $c(v,u) = 0$, și $f(u,v) = -f(v,u) = 29$, atunci arcul (u,v) are $40 - 29 = 11$ unități de capacitate reziduală, iar arcul (v,u) are $0 - (-29) = 29$ unități de capacitate reziduală.

Pe scurt, problema fluxului maxim este o problemă clasică de optimizare în teoria grafurilor, care implică găsirea cantității maxime de flux ce poate fi trimis printr-o rețea de conducte, canale sau alte căi, sub rezerva constrângerilor de capacitate. Problema poate fi utilizată pentru a modela o mare varietate de situații din lumea reală, cum ar fi sistemele de transport, rețelele de comunicații sau alocarea resurselor.

O abordare comună pentru rezolvarea problemei fluxului maxim este algoritmul Ford-Fulkerson (Ford și Fulkerson 1956), care se bazează pe ideea drumurilor de mărire. Acest algoritm are ca parametri de intrare o rețea antisimetrică, $G = (V, E, c, s, t)$, cu nodul sursă, s, și nodul stoc, t, iar ca parametru de ieșire este f, fluxul maxim admisibil prin rețeaua G.

4.2. Problema fluxului maxim generalizat

Problema fluxului maxim generalizat extinde problema tradițională a fluxului maxim permițând ca fluxul să se modifice în timp ce este trimis prin rețea. Ca și înainte, fiecare arc (u, v) are o capacitate c(u, v) care limitează cantitatea de flux trimis în acel arc. În plus, fiecare arc (u, v) are asociat un coeficient pozitiv $\alpha(u,v) > 0$, numit factor de câștig / pierdere. Factorul de câștig / pierdere este o funcție $\alpha : E \to \mathbb{R}_{>0}$. Pentru fiecare unitate de flux care intră în arcul (u,v) la prin nodul u, doar $\alpha\ (u,v)$ unități ajung la nodul v. Un arc cu pierderi este un arc cu $\alpha < 1$, iar arcul pentru care există câștig are $\alpha > 1$. Fără a pierde generalitatea, presupunem că funcția de câștig / pierdere este simetrică, adică $\alpha(u,v) = 1/\alpha(v,u)$. Dacă această presupunere nu este satisfăcută, putem adăuga arcul simetric și îi putem da o capacitate de zero (Wayne 1999).

Pentru a rezolva problema generalizată a fluxului maxim pentru determinarea fluxului cu pierdere minimă, propunem o adaptare a algoritmului Ford – Fulkerson (**Algoritmul 4.**), astfel încât, la fiecare iterație, pentru găsirea unui nou drum în rețeaua reziduală se aplică **Algoritmul 3.2** de determinare a drumului cu pierderea minimă. Alegerea **Algoritmul 3.2** se explică prin faptul că, atât în cazul unei rețelelor care au doar pierderi pe arce, cât și în cazul rețelelor cu pierderi sau câștiguri pe arce, în rețeaua reziduală rezultată se află ambele tipuri de arce din cauza modului de calcul a factorului de pierdere în respectiva rețea. Cu alte cuvinte, dacă în rețeaua inițială un arc (u,v) are factorul de câștig / pierdere $\alpha(u,v)$, atunci factorul de câștig / pierdere în rețeaua reziduală este $\alpha_f(u,v) = \alpha(u,v)$ pe arcul direct, iar pe arcul invers avem $\alpha_f(v,u) = \frac{1}{\alpha(u,v)}$.

Algoritmul 4.1 Adaptare a algoritmului Ford – Fulkerson pentru determinarea fluxului maxim într-o rețea generalizată

Parametri de intrare:

- O rețea $G = (V, E, s, t, c, \alpha)$
- Nodul sursă, s
- Nodul stoc, t

Parametri de ieșire:

- f – fluxul cu pierderi minime în G

Se consideră un flux admisibil, $f = 0$
Se inițializează rețeaua reziduală $G_f = G$
Pentru fiecare arc $(u, v) \in E$
$\alpha_f(u, v) = \alpha(u, v)$
$\alpha_f(v, u) = \frac{1}{\alpha(u,v)}$
Sfârșit pentru
Cât timp există drum de mărire de s la t în G_f
Se găsește un drum de mărire, P, în G_f folosind **Algoritmul 3.2**
Se actualizează rețeaua reziduală, G_f folosind **Algoritmul 4.**
Sfârșit cât timp

Pentru actualizarea rețelei reziduale, G_f, de-a lungul drumului, P, găsit se folosește **Algoritmul 4..** Acest algoritm implică determinarea fluxului admisibil maxim pe drumul de mărire și apoi actualizarea capacităților reziduale și a factorilor de câștig / pierdere în rețeaua reziduală pentru a reflecta acest flux. Astfel, rețeaua reziduală este pregătită pentru iterații ulterioare ale algoritmului de flux. Algoritmul are două etape:

1. Determinarea flux admisibil pe drumul de mărire P:
   - Se inițializează fluxul f cu capacitatea reziduală a primului arc (s, u) din P.
   - Pentru fiecare arc (u, v) din drumul P cât timp nodul u nu este destinația t:
     - Se actualizează fluxul f cu minimul dintre fluxul curent $f \cdot \alpha_f(u, v)$, și capacitatea reziduală, $c_f(u, v) \cdot \alpha_f(u, v)$.
     - Se avansează la următorul nod $u = v$ de pe drumul P.
2. Actualizare rețea reziduală, G_f
   - Se pornește de la nodul stoc, $v = t$.
   - Pentru fiecare arc (u, v) de pe drumul P, cât timp nodul v este sursa s:
     - Se ajustează fluxul f în funcție de factorul de câștig / pierdere $\alpha_f(u, v)$.
     - Pe arcul (u, v) se actualizează capacitatea reziduală $c_f(u, v) = c_f(u, v) - f/\alpha_f(u, v)$, iar pentru arcul invers se adaugă fluxul la capacitatea sa reziduală $c_f(v, u) = c_f(v, u) + f$.
     - Se avansează pe drumul P, către sursă, la nodul anterior $v = u$.

Algoritmul 4.2 Actualizarea rețelei reziduale după găsirea unui nou drum de mărire în G_f

Parametri de intrare:

- Rețeaua reziduală $G_f = (V, E, s, t, c_f, \alpha_f)$
- Drumul de mărire de la s la t găsit, P în G_f

Parametri de ieșire:

- Rețeaua reziduală actualizată, G_f
- Fluxul admisibil, f

//determinare valoare flux în nodul t pe drumul P
Se consideră un flux, $f = c_f(s, u)$, unde $(s, u) \in P$ este primul arc din P

$u = s$
Cât timp $u \neq t$
Se consideră arcul $(u, v) \in P$
$f = min\{f \cdot \alpha_f(u, v),\ c_f(u, v) \cdot \alpha_f(u, v)\}$
$u = v$
sfârșit cât timp
// Actualizare G_f
$v = t$
Cât timp $v \neq s$
Se consideră arcul $(u, v) \in P$
$c_f(u, v) = c_f(u, v) - f/\alpha_f(u, v)$
$c_f(v, u) = c_f(v, u) + f$
$f = f/\alpha_f(u, v)$
$v = u$
sfârșit cât timp

4.3. *Rezultate și discuții*

Pentru implementarea algoritmului propus rezultatele din punctul de vedere al accelerației atunci când se utilizează programarea GPU sunt similare cu cele obținute pentru **Algoritmul 3.2** întrucât eficientizarea se poate realiza prin utilizarea algoritmului Bellman - Ford paralel, pentru fiecare determinare de drum cu pierdere minimă. Astfel, după cum se poate observa în **Algoritmul 4.** complexitatea acestuia este dată de complexitatea **Algoritmul 3.2** înmulțită cu numărul de iterații în care se determină câte un drum de mărire.

Eficiența GPU crește inițial cu complexitatea problemei, dar după un anumit punct începe să scadă, posibil din cauza saturației resurselor GPU. Speed-up-ul variază între 1.04 și 5.72, cu valori maxime în cazul problemelor de dimensiuni medii. Performanța GPU este semnificativ superioară CPU pentru majoritatea configurațiilor testate, mai ales pentru dimensiuni medii și mari ale grafurilor.

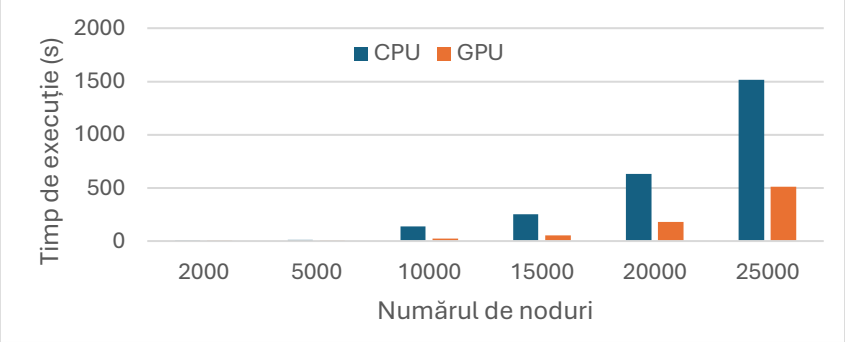


Figura 4.1 Timpi de execuție pentru **Algoritmul 4.** în rețele dense

După cum se poate observa în **Figura 4.1** timpul de execuție pe GPU este semnificativ mai scurt decât CPU pentru toate rețelele dense analizate, indiferent de numărul de noduri. Eficiența GPU este evidentă, mai ales pentru rețele dense de dimensiuni mari. Astfel, timpul de execuție pe GPU crește mai lent comparativ cu CPU pe măsură ce numărul de noduri crește. Raportul de speed-up scade pe măsură ce numărul de noduri crește. Pentru noduri mai mici (2000-10000), GPU oferă un speed-up considerabil (de 2.7-5.7 ori). Pentru noduri mai mari (15000-25000), acest raport scade ușor, dar rămâne semnificativ (2.98-4.45 ori).

Capitolul 5. Interpolare rapidă pe GPU pentru generare de hărți

Acest capitol se bazează pe lucrările (**Spridon**, Deaconu și Ciupala, ICCSA 2023) și (Ciupala, Deaconu și **Spridon** 2021), la care sunt coautor. În capitolul curent prezint metode GPU pentru obținerea de hărți de poluare și geomagnetice prin folosind metode de interpolare, pornind de la măsurători în diferite puncte dintr-o anumită zonă geografică. Am realizat analize de precizie a hărților obținute și de eficiență a metodelor GPU utilizate, rezultatele fiind prezentate în cele ce urmează.

5.1. Metode de interpolare bi-dimensională

Formularea generală a problemei de interpolare spațială poate fi definită după cum urmează:

Având în vedere n valori ale unui fenomen studiat $V(j)$, cu $j = \overline{1,n}$, măsurate în poziții discrete de coordonate $r_j = (x_j, y_j)$, în cazul unui spațiu bidimensional, se caută o funcție $F(r)$ care să îndeplinească condițiile:

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

Deoarece există un număr infinit de funcții care îndeplinesc această cerință, trebuie impuse condiții suplimentare, care definesc caracterul diferitelor tehnici de interpolare. Exemple tipice sunt condițiile bazate pe concepte geostatistice (kriging), localizare (metodele celui mai apropiat vecin și elementele finite), netezimea și spline sau formele funcționale ad-hoc (polinoame, multi-quadric). Alegerea condiției suplimentare depinde de caracterul fenomenului modelat și de tipul aplicației.

Există mai multe metode de interpolare utilizate pentru obținerea de hărți în domenii precum cartografie, geografie și analiza datelor spațiale. În cele ce urmează voi descrie două dintre cele mai recente utilizate astfel de metode.

- *Metoda IDW (Inverse Distance Weighting)* presupune că valoarea estimată este o funcție de distanța dintre punctul în care se face estimarea și locațiile eșantioanelor, astfel încât valorile măsurate aflate mai apropiate de poziția în care se află punctul de estimat au o influență mai mare asupra valorii estimate decât cele mai îndepărtate.
- Kriging presupune estimarea valoarii necunoscute $z(u)$ într-o anumită locație, u, pe baza unei medii ponderate a valorilor observate, $V(r_i)$, în locațiile punctelor eșantion din apropiere, r_i . Ponderile sunt alese astfel încât să se minimizeze eroarea de estimare și sunt determinate pe baza structurii de corelare spațială a datelor. Ponderile kriging sunt obținute prin rezolvarea unui sistem de ecuații liniare care exprimă funcția de autocovarianță spațială a datelor.

5.2. Accelerarea metodelor de interpolare folosind CUDA

Algoritmii implementați au fost testați pe un sistem cu procesor Intel(R) Core(TM) i7-10750H @ 2.60GHz 2.59 GHz, 16.0 GB RAM, GPU NVIDIA GeForce RTX 2060 și sistemul de operare Windows 10 Pro. Aceste metode de interpolare au fost accelerate prin utilizarea programării CUDA pentru obținerea de hărți de rezoluție mare în timp real. Acest instrument ar putea fi folosit, de exemplu, pentru monitorizarea modificărilor geomagnetice ale unei suprafețe mari pentru a identifica

modificările care ar putea avea loc în structura Pământului sau pentru identificarea regiunilor cu anumite proprietăți magnetice ori pentru monitorizarea în timp real a hărților de poluare din diferite zone.

Pseudocodul algoritmului IDW este prezentat mai jos (Ciupala, Deaconu și **Spridon** 2021).

Algoritmul 5.1 Algoritmul IDW

Parametri de intrare: $p,\ x_{min}, x_{max}, y_{min}, y_{max}\ , n, m;$

/ determinare rezoluției pe cele două direcții: x și y */*

$dx = \frac{x_{max} - x_{min}}{n}$

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

/ calcularea valorilor estimate în fiecare punct al gridului */*

$y = y_{min}$

Pentru $i = 1, n$ **execută**

$x = x_{min}$

Pentru $j = 1, m$ **execută**

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

$x = x + dx$

sfârșit pentru

$y = y + dy$

sfârșit pentru

unde g_{ij} sunt punctele unui grid 2D de $m \times n$ valori interpolate, $m, n \in N^*$ pentru o regiune dreptunghiulară dată de coordonatele x_{min}, x_{max}, y_{min}, $y_{max} \in R$, $(x_{min} < x_{max}, y_{min} < y_{max})$. Astfel, **Algoritmul 5.1** creează un grid 2D pe o suprafață delimitată de coordonatele x_{min}, x_{max}, y_{min} , y_{max} . dx și dy sunt calculate pentru a determina distanța dintre punctele gridului pe axele x și, respectiv, y. Ulterior, se parcurg toate punctele gridului și se calculează valoarea g_{ij} în fiecare punct de coordonate (x, y) utilizând o media ponderată $V(x, y)$, iar distanța dintre două puncte pe grid a fost calculată folosind formula pentru distanța pe Pământ între două puncte de anumite coordonate GPS.

Pentru a folosi CUDA pentru IDW, mai întâi trebuie să paralelizăm algoritmul. În IDW, trebuie să calculăm distanța dintre punctele în care se dorește estimare și fiecare dintre punctele eșantion. Acest calcul al distanței poate fi paralelizat prin alocarea fiecărui fir de execuție GPU unui singur punct al gridului și calculând distanțele către toate punctele eșantion.

După calcularea distanțelor, se calculează ponderile pentru fiecare punct eșantion pe baza distanței până la punctul de estimat. Acest calcul al ponderilor poate fi, de asemenea, paralelizat prin alocarea fiecărui fir de execuție GPU unui singur punct eșantion și calculând ponderea acestuia pentru toate punctele în care se dorește estimarea valorii.

În cele din urmă, putem folosi ponderile calculate pentru a interpola valorile în punctele gridului. Acest pas de interpolare poate fi, de asemenea, paralelizat prin alocarea fiecărui fir de execuție GPU unui singur punct de estimat și calculând valoarea acestuia pe baza mediei ponderate a valorilor punctelor eșantion.

Algoritmul de interpolare kriging a fost implementat urmând 4 pași **(Algoritmul 5.2)**

Algoritmul 5.2 Algoritmul kriging

Parametri de intrare: x_{min}, x_{max}, y_{min}, y_{max} , n, m, v;

Calculul punctelor de semi-variantă
Calculul coeficienților de semi-variantă folosind metoda celor mai mici pătrate
Calculul ponderilor de interpolare
Calculul valorilor interpolate

Pentru paralelizarea algoritmului de kriging folosind CUDA, sunt necesari mai mulți pași: calculul variogramei, calculul matricei de kriging și calculul ponderii kriging.

Calculul variogramei implică calcularea semi-variantei dintre toate perechile de puncte eșantion. Acest pas poate fi paralelizat prin atribuirea fiecărui fir GPU unei singure perechi de puncte eșantion și calculând semi-varianța acestora.

Calculul matricei de kriging implică inversarea unei matrice care depinde de semi-variantele dintre punctele eșantion.

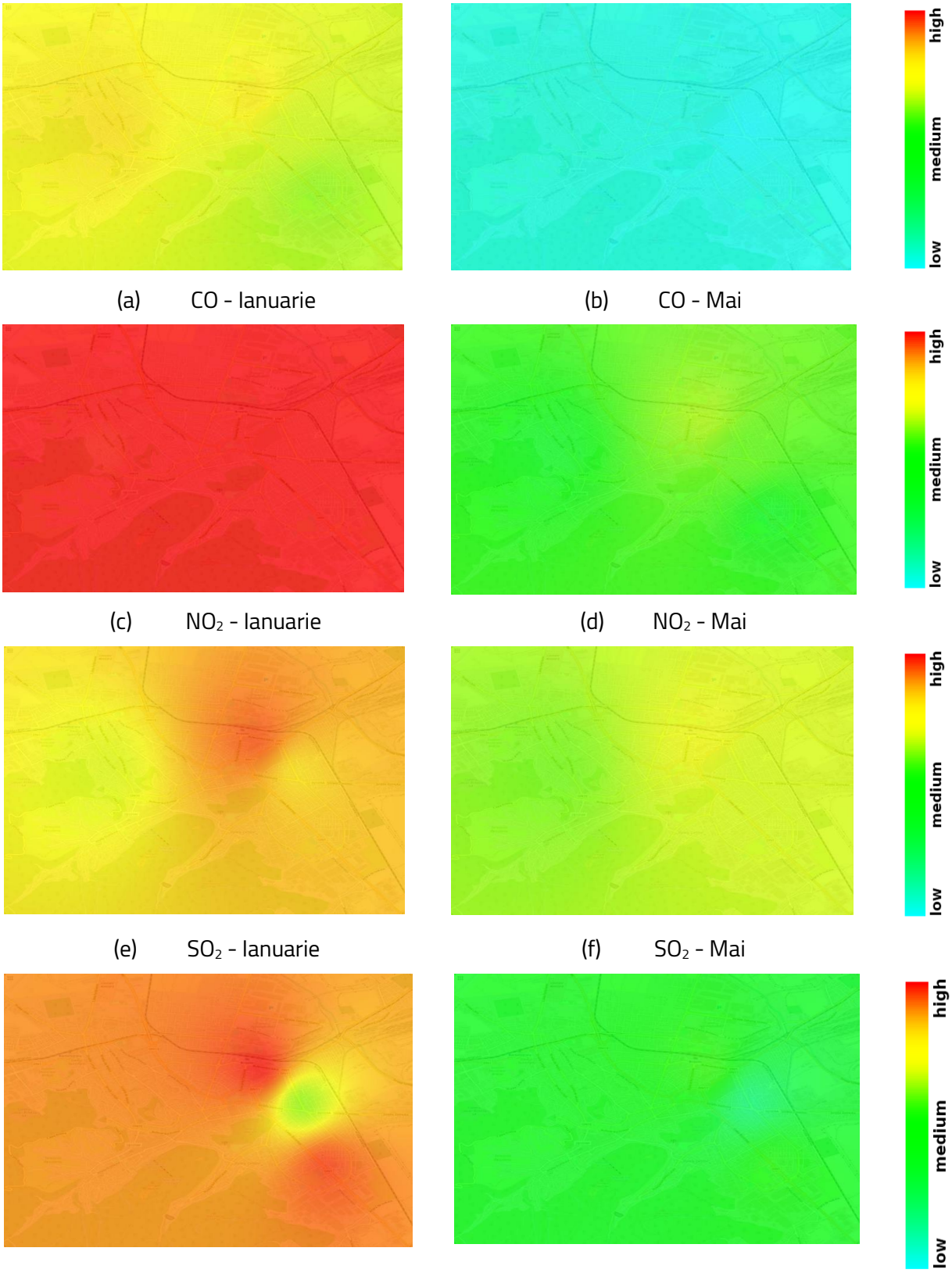
În cele din urmă, calculul ponderilor kriging implică calcularea ponderilor pentru fiecare punct eșantion pe baza distanței sale și a corelației spațiale cu punctul de estimat. Acest pas poate fi paralelizat prin atribuirea fiecărui fir de execuție GPU unui singur punct de estimat și calculând ponderile acestuia pentru toate punctele eșantion.

5.3. Studiul hărților de poluare a Brașovului pe perioada pandemiei

Am utilizat metoda IDW pentru a crea hărți de poluare (grile) pentru zona urbană a Brașovului și a trage concluzii privind poluarea pentru anul 2020. De asemenea, am făcut o comparație a calității aerului în perioada de blocare (majoritatea activităților economice și sociale au fost oprite) din cauza pandemiei de Covid-19 și perioada în care economia a fost repornită. Pentru acest studiu, au fost luate în considerare concentrațiile de monoxid de carbon (CO), dioxid de sulf (SO_2), dioxid de azot (NO_2) și particule (PM10). Datele pentru cele patru stații care raportează poluarea din oră în oră din Brașov au fost descărcate de la (Rețeaua Națională de Monitorizare a Calității Aerului 2021) pentru prima jumătate a anului 2020 pentru CO, PM10, SO_2 și NO_2.

Folosind algoritmul IDW, am generat hărți ale concentrațiilor poluanților din oră în oră, hărți cu mediile pe 24 de ore pentru fiecare poluant, hărți cu mediile concentrațiilor pe o lună și hărți cu mediile concentrațiilor pentru fiecare zi a săptămânii pentru a vedea cum diferă poluarea in zilele lucrătoare fata de zilele de sfârșit de săptămână. De asemenea, am comparat grafic statisticile pe zile din săptămână și am urmărit evoluția poluării pe lună. Acest lucru a fost făcut separat pentru fiecare dintre cele patru stații și în medie pentru toate stațiile.

Pentru fiecare poluant am creat două hărți pentru a compara concentrația medie în ianuarie (înainte de lock-down) și mai (ultima lună de lock-down) (vezi **Figura 5.1**). Comparând cele două imagini, este evident că calitatea aerului a fost îmbunătățită semnificativ pentru fiecare dintre factorii poluanți datorită activității industriale reduse și a numărului scăzut de vehicule care au circulat în perioada respectivă.



(a) CO - Ianuarie (b) CO - Mai

(c) NO_2 - Ianuarie (d) NO_2 - Mai

(e) SO_2 - Ianuarie (f) SO_2 - Mai

(g) PM10 - Ianuarie (h) PM10 - Mai

Figura 5.1 *Comparația mediilor concentrațiilor principalilor poluanți pentru lunile Ianuarie 2020 (a, c, e, g) și Mai 2020 (b, d, f, h)*

Rezultatele experimentale prezentate în **Tabel 5.** au arătat că implementările bazate pe CUDA care rulează pe GPU au condus la o creștere a vitezei de execuție în funcție de rezoluția imaginii.

Interpolarea cu IDW a fost folosita pentru a obține imagini cu dimensiuni cuprinse între 150 x 100 și 4800 x 3200. Experimentele au arătat că pentru imaginile mici (150 x 100 și 300 x 200) timpul CPU a fost mai bun. Pentru imaginile mari, accelerarea GPU-ului a fost constantă (de până la 19 ori mai rapidă).

Tabel 5.1 *Studiul vitezei de execuție pe GPU*

Dimensiunea imaginii	Timpul de execuție pe CPU (s)	Timpul de execuție pe GPU (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. Studiul hărților geomagnetice ale României

Datele și hărțile geomagnetice sunt instrumente esențiale pentru înțelegerea câmpului magnetic al Pământului și a diverselor aplicații ale acestuia. Datele geomagnetice oferă perspective asupra structurii și dinamicii interiorului Pământului, în timp ce hărțile geomagnetice sunt folosite pentru navigație, cartografiere geologică și cercetare științifică. Datele și hărțile geomagnetice au aplicații practice în industrie și întreprinderi comerciale, în special în explorarea minerală, dezvoltarea energiei și navigație.

5.5. Metode CUDA pentru obținerea de hărți geomagnetice

Datele geomagnetice pentru obținerea hărții geomagnetice a României, utilizând metode de interpolare IDW și kriging au fost obținute de la stații geomagnetice românești și cu ajutorul aplicației Physics Toolbox Sensor Suite în peste 1300 de poziții GPS răspândite în toată țară. Datele au fost culese cu ajutorul acțiunii Citizen Science din proiectul European Researchers Night 2018-2019 Handle with Science, finanțat din H2020, AG nr. 818795/2018.

Regiunea studiată se află între 21° lon E și 29° lon E și între 41° lat N și 49° lat N. Grilele obținute au rezoluții 400 x 400, 800 x 800, 1200 x 1200 și 1600 x 1600, deci fiecare punct al grilei are aproximativ 2 km, 1 km, 0,75 km și, respectiv, 0,5 km. În **Figura 5.** și **Figura 5.** sunt prezentate hărți geomagnetice cu rezoluție de 1 km pentru regiunea României obținute prin IDW și, respectiv, interpolarea kriging.

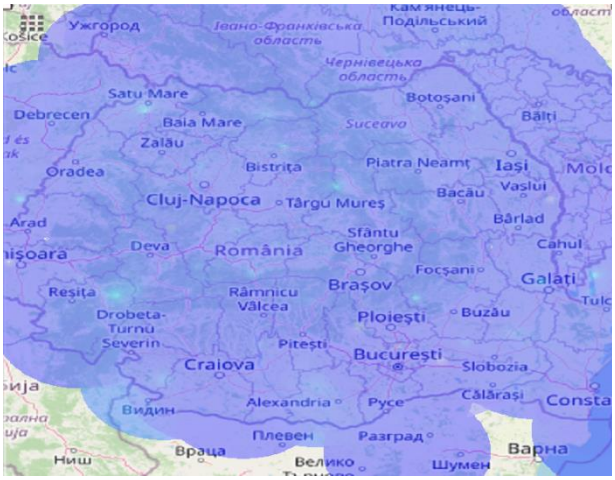


Figura 5.2 *Harta geomagnetică obținută prin IDW*

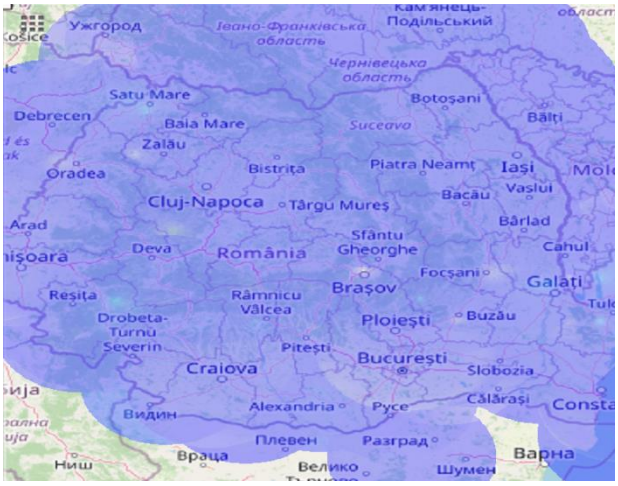


Figura 5.3 *Harta geomagnetică obținută prin interpolare kriging*

Rezultatele au acuratețe o mai bună pentru metoda de interpolare kriging, pentru toate rezoluțiile studiate. De exemplu, dacă pentru IWD valoare mediană a erorii este între 4,476 și 4,895 μT, în funcție de rezoluție, pentru metoda de a kriging această valoare între 2,871 și 3,687 μT. Mai mult, în **Figura 5.**, pot fi observate valorile mai mici ale erorii medii a câmpului geomagnetic pentru metoda kriging.

Figura 5.4 *Comparație între eroarea medie pentru câmpul geomagnetic obținut prin IDW și Kriging*

Cu alte cuvinte, comparând rezultatele, putem observa că metoda kriging de interpolare are valori mai mici pentru toate erorile analizate față de IDW, ceea ce indică o performanță mai bună a metodei acesteia pentru interpolarea datelor de geomagnetism. De asemenea, în general, cu cât este mai mare rezoluția gridului, cu atât valorile erorilor obținute sunt mai mici, indicând o mai bună acuratețe a interpolării odată cu creșterea rezoluției.

Calculul complex din metoda kriging duce la un timp de execuție crescut pentru toate rezoluțiile, în comparație cu IDW **Figura 5.5**. Viteza obținută pentru implementarea IDW este foarte mare și crește odată cu rezoluția grilei, de până la 104 ori pentru grila 1600x1600. Deși, viteza pentru kriging nu este la fel de mare ca pentru IDW **Figura 5.6**, pentru cea mai mare rezoluție studiată, timpul de execuție la rularea pe GPU a scăzut de 10 ori comparativ cu CPU. Astfel, se observă că timpul de execuție pentru ambele metode, IDW și kriging, este semnificativ mai mic pentru implementările pe GPU.

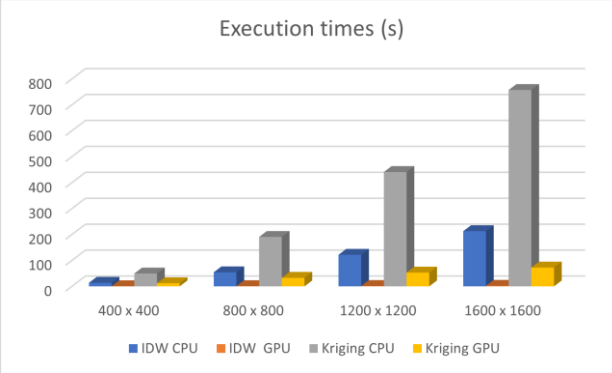


Figura 5.5 *Timpii de execuție pentru IDW și Kriging pe CPU și GPU*

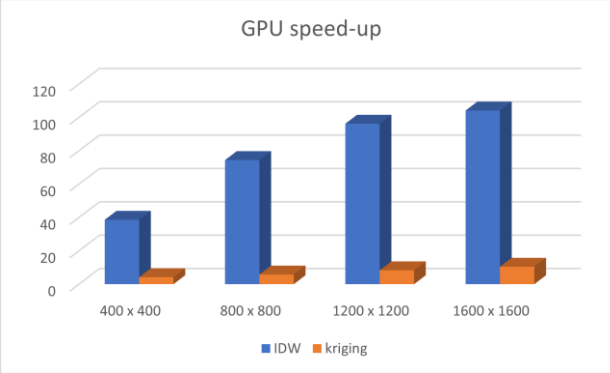


Figura 5.6 *Speed-up-ul pentru IDW și Kriging*

Se observă că timpul de execuție pe GPU este semnificativ mai mic decât timpul de execuție pe CPU pentru ambele metode de interpolare și pentru toate oricare dimensiune a gridului. Acest lucru indică faptul că paralelizarea algoritmilor de interpolare folosind CUDA a dus la o accelerare semnificativă a procesului de interpolare. Mai precis, speed-up-ul crește odată cu mărirea dimensiunii gridului, ceea ce arată că beneficiile paralelizării sunt mai pronunțate pentru seturi de date mai mari. Astfel, implementarea algoritmilor de interpolare pe GPU folosind CUDA poate fi o alegere eficientă pentru îmbunătățirea performanței și timpului de execuție al acestor algoritmi. Comparând speed-up-ul pentru cele doua metode, se observă că, în general, speed-up-ul pentru Kriging este mai mic decât cel pentru IDW la oricare dintre dimensiunile analizate ale gridului. Acest lucru sugerează că paralelizarea algoritmului de interpolare Kriging pe GPU folosind CUDA aduce beneficii mai mici în comparație cu paralelizarea algoritmului IDW. Cu toate acestea, ambele metode pot fi eficientizate în mod clar prin utilizarea GPU-ului pentru creșterea accelerației, iar diferența în speed-up este influențată de natura specifică a algoritmilor și de modul în care aceștia sunt paralelizați. Totuși, dacă se iau în considerare atât performanța de execuție, cât și acuratețea rezultatelor, putem concluziona că, deși IDW oferă un speed-up mai mare și un timp de execuție mai mic, Kriging este o opțiune mai bună atunci când se urmărește obținerea unor rezultate cu precizie ridicată în detrimentul unui timp de execuție mai mic.

Capitolul 6. Concluzii și perspective

În lucrarea de față în **Capitolul 1** au fost prezentate o serie de avantaje și dezavantaje ale programării pe GPU și s-a făcut o trecere în revistă a câtorva dintre cele mai importante aplicații ale programării pe GPU. Aceste informații au fost publicate în lucrarea (**Spridon**, Advances in CUDA for computational physics, 2023).

Capitolele următoare prezintă câteva dintre rezultatele personale publicate în reviste științifice sau prezentate la conferințe internaționale. Astfel, **Capitolul 2** prezintă un algoritm rapid și de încredere numit AGRFA pentru a genera rețele aleatoare. Rețelele rezultate pot fi utilizate pentru a testa corectitudinea și eficiența algoritmilor dezvoltați pentru probleme de flux de rețea, de exemplu, fluxul de cost minim, fluxul maxim sau probleme de flux cu mai multe mărfuri. Versiunea paralelizată CUDA a AGRFA s-a dovedit a fi de până la 19 ori mai rapidă atunci când trebuie generate rețele de dimensiuni mari. Având în vedere evoluțiile ulterioare, ar putea fi identificate și alte probleme în rețele specifice în

care AGRFA poate fi adaptat. Aceste rezultate au fost publicate în calitate de coautor în lucrarea (Deaconu și **Spridon** 2021).

În **Error! Reference source not found.**, introduc și rezolv o problemă practică numită problema drumului cu pierdere minimă sau drumul cu rata de livrare maximă. Aceasta problemă constă în găsirea drumului de la un nod sursa la un alt nod dat t numit s stoc (sink) într-o rețea generalizată, care are un cu un factor câștig/pierdere atașat fiecărui arc, astfel încât pierderea să fie minimă între toate drumurile $s - t$. Rezultatele arată o viteză mare atunci când se utilizează programarea GPU pentru **Algoritmul 3.1** pentru rețele mari și dense. O îmbunătățire a timpului de execuție a fost obținută și pentru **Algoritmul 3.2**, folosind algoritmul Bellman-Ford în implementarea bazată pe GPU. Rezultatele au fost prezentate în lucrarea (Deaconu, **Spridon** și Ciupala 2023).

Capitolul 4 prezintă o aplicație pentru determinarea drumului cu pierderi minime Astfel, algoritmul MLPP, este utilizat într-o rețea generalizată pentru determinarea fluxului minim. Este propusă o adaptare a algoritmului lui Ford -Fulkeson, în care, la fiecare iterație este căutat drumul cu pierderi minime. Se obține astfel fluxul cu pierderea minimă în rețeaua respectivă.

În **Capitolul 5** se descrie obținerea de hărți georeferențiate cu ajutorul metodelor de interpolare bidimensionale și pornind de la valori măsurate în puncte discrete, într-o anumită zonă geografică. Astfel, s-au studiat hărțile de poluare ale Brașovului din perioada lock-down-ului din cauza COVID-19. Hărțile au fost obținute folosind metoda de interpolare IDW și pentru rezoluții mari ale acestora s-a utilizat CUDA obținându-se creșteri semnificative ale vitezei de execuție. Totodată s-a studiat obținerea de hărți geomagnetice ale României folosind metodele de interpolare IDW și kriging și investigând atât acuratețea hărților obținute cât și viteza de obținere a acestora. Eroarea estimărilor din hărțile geomagnetice este mai mică în cazul interpolării prin metoda kriging, iar viteza de execuție a fost demonstrat că poate fi îmbunătățită cu ajutorul programării pe GPU folosind CUDA. Lucrările care au stat la baza acestui capitol sunt (Ciupala, Deaconu și **Spridon** 2021) și (**Spridon**, Deaconu și Ciupala, ICCSA 2023).

Ca perspective viitoare de cercetare, în domeniul interpolării bidimensionale doresc să studiez și alți paralelizarea pe GPU a altor algoritmi de interpolare pe seturi de date neregulate, distribuite neuniform și obținerea de hărți de rezoluție înaltă.

De asemenea, doresc să dezvolt modele hibride care combină puterea de procesare a GPU-urilor cu metode de paralelizare pe CPU pentru creșterea vitezei de execuție a algoritmilor atunci când se aplica pe rețele de mari dimensiuni. Este necesară pentru aceasta studierea și evaluarea performanței algoritmilor paraleli în contextul teoriei grafurilor pe GPU și de asemenea, identificarea limitărilor și optimizarea codului pentru a profita la maxim de arhitectura CUDA.

În plus, doresc să folosesc metode GPU pentru accelerarea calculului și rezolvarea unor problemelor complexe din domeniul fizicii computaționale, în special în ceea ce privește accelerarea simulărilor care modelează transportul energiei, particulele și interacțiunile în interiorul plasmei și dezvoltarea de algoritmi pe GPU pentru a analiza fenomene complexe, cum ar fi transportul turbulent în plasme.

Lucrări publicate în domeniul tezei

Lucrări publicate în reviste cu factor de impact ISI:

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

Lucrări publicate în reviste indexate Scopus:

1. **Spridon, D.** 2023. "Advances in CUDA for computational physics", *Bulletin of the Transilvania University of Brașov. 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 Brașov. Series III: Mathematics and Computer Science*, 63(1), 247-256.

Lucrări prezentate și publicate în proceedings-urile unor conferințe internaționale indexate 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

Bibliografie selectivă

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

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

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

Ciupala, L., A. Deaconu, și D. **Spridon**. „IDW map builder and statistics of air pollution in Brasov." *Bulletin of the Transilvania University of Brasov*, 2021: 247-256.

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

Deaconu, A., și E. Ciurea. „The inverse maximum flow problem under Lk norms." *Carpathian Journal of Mathematics* 28 (2012): 59–66.

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

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

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

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

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

Ford, L.R., și D. R. Fulkerson. „Maximal flow through a network." *Canadian Journal of Mathematics*, 1956: 399–404.

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

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

Huang, J. *NVIDIA.* October 2023. https://s201.q4cdn.com/141608511/files/doc_presentations/2023/Oct/01/ndr_presentation_oct_2023_final.pdf.

Marinescu, C., A. Deaconu, E. Ciurea, și D. Marinescu. „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, 2010. 1251–1256.

—. „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, 2010. 1245–1250.

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

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

Rețeaua Națională de Monitorizare a Calității Aerului. *https://www.calitateaer.ro.* 2021. https://www.calitateaer.ro.

Schrijver, A. „On the history of the shortest path problem." *Documenta Mathematica, Extra Volume ISMP*, 2012: 155–167.

Spridon, D. „Advances in CUDA for computational physics." *Bulletin of the Transilvania University of Brasov* 3(65), nr. 2 (2023): 227-236.

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

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

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

Tayyebi, J., și A.M. Deaconu. „Inverse Generalized Maximum Flow Problems." *Mathematics*, 2019: 899.

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