Delia Elena SPRIDON

Aplicații avansate ale programării pe GPU

Editura Universității Transilvania din Brașov

2024

EDITURA UNIVERSITĂȚII TRANSILVANIA DIN BRAȘOV

Adresa: Str. Iuliu Maniu nr. 41A
500091 Brașov
Tel.: 0268 476 050
Fax: 0268 476 051
E-mail: editura@unitbv.ro

Editură recunoscută CNCSIS, cod 81

ISBN 978-606-19-1759-4 (ebook)

Referenți științifici:

Prof. dr. Petrică POP SITAR, Universitatea Tehnică din Cluj-Napoca
Prof. dr. Ioan TOMESCU, Universitatea din București

Cuprins

Introducere

Această carte se bazează pe o cercetare amănunțită și include, în esență, teza de doctorat a autoarei, intitulată: „*Metode GPU pentru creșterea performanței computaționale în teoria grafurilor și generarea hărților*".

Odată cu evoluția tehnologiei și a sistemelor de calcul, acestea din urmă au devenit indispensabile în aproape orice domeniu științific: economic, social etc. Calculatoarele de astăzi ne permit simularea și rezolvarea de probleme foarte complexe. Astfel, în era digitală actuală, cu cerințele tot mai complexe ale aplicațiilor moderne, creșterea performanței în domeniul calculului paralel a devenit o prioritate esențială. O soluție pentru atingerea acestui obiectiv este reprezentată de utilizarea metodelor GPU (Graphics Processing Unit) pentru accelerarea operațiilor critice în aplicații practice. Acest lucru se datorează faptului că, deși GPU-urile au fost inițial dezvoltate pentru procesări grafice, în timp, acestea și-au extins în mod semnificativ sfera de aplicare, devenind instrumente esențiale în diverse domenii precum inteligența artificială, simulări științifice, randare 3D și multe altele.

Diferența majoră între unitățile de procesare tradiționale și GPU-uri constă în abordarea paralelă a sarcinilor de lucru. În timp ce procesoarele tradiționale se axează pe execuția secvențială a instrucțiunilor, GPU-urile sunt proiectate pentru a gestiona simultan un număr crescut de sarcini prin intermediul unui mare număr de nuclee de procesare. Această arhitectură paralelă conferă GPU-urilor o capacitate impresionantă de accelerare, permițând rezolvarea rapidă a problemelor complexe. Astfel, programarea GPU implică utilizarea unităților de procesare grafică pentru a efectua operații de calcul cu scop general. GPU-urile sunt capabile să efectueze un număr mare de calcule simultan datorită numărului mare de core-uri ce pot fi încorporate pe plăcile video.

Metodele GPU pot îmbunătăți semnificativ anumite aplicații practice atât din punct de vedere al performanței, cât și din punct de vedere al eficienței energetice. De exemplu, în domeniul inteligenței artificiale, antrenarea și inferența modelelor de învățare profundă sunt exemple evidente de aplicații care profită de

puterea de procesare paralelă masivă a GPU-urilor. De asemenea, în domeniul simulărilor științifice, precum fizica particulelor sau dinamica fluidelor, GPU-urile permit rezolvarea rapidă a problemelor complexe, deschizând noi orizonturi în cercetare.

Cu toate acestea, integrarea eficientă a metodelor GPU în aplicații practice necesită expertiză și adaptabilitate. Dezvoltatorii trebuie să optimizeze codul pentru a beneficia la maximum de arhitectura paralelă a GPU-urilor și să gestioneze eficient transferul de date între CPU (Central Processinf Unit) și GPU. În același timp, selectarea adecvată a algoritmilor și a aplicațiilor care pot beneficia în mod semnificativ de accelerarea GPU-urilor reprezintă o etapă crucială în implementarea cu succes a acestor tehnologii.

Pe scurt, metodele GPU pentru creșterea performanței în aplicații practice au devenit un element esențial în arsenalul dezvoltatorilor din diverse domenii. Cu o putere de procesare excepțională și o adaptabilitate în creștere, GPU-urile deschid noi orizonturi în ceea ce privește rezolvarea problemelor complexe și accelerarea evoluției tehnologice.

CUDA (Compute Unified Device Architecture) este o platformă de calcul paralelă și un model de programare dezvoltat de NVIDIA (corporație multinațională și companie tehnologică americană) pentru programarea pe GPU. Aceasta oferă un set de instrumente și API-uri care permit dezvoltatorilor să scrie cod paralel care poate fi executat pe plăcile grafice concepute de NVIDIA.

În ultimii ani, a crescut din ce în ce mai mult interesul pentru utilizarea programării pe GPU utilizând CUDA pentru procesarea grafurilor. Acest interes a fost determinat de nevoia de a procesa grafuri de mari dimensiuni, rapid și eficient, precum și de disponibilitatea GPU-urilor puternice care pot efectua calcule complexe la o viteză mare. În aceasta lucrare, sunt descrise câteva dintre rezultatele cercetării recente în ce privește programarea GPU cu ajutorul CUDA în teoria grafurilor, incluzând rezultatele proprii ale autoarei.

Teoria grafurilor este o ramură a matematicii ce se ocupă cu studiul grafurilor, care sunt structuri matematice folosite pentru a modela relațiile dintre obiecte. Teoria grafurilor are aplicații într-o gamă largă de domenii, inclusiv informatică, fizică, biologie, științe sociale. Studiul grafurilor de mari dimensiuni a devenit din ce în ce mai important în ultimii ani datorită prevalenței cantităților

mari de date și a necesității de a procesa și analiza seturi mari de date rapid și eficient.

Una dintre provocările majore în procesarea grafurilor de mari dimensiuni este complexitatea de calcul a multor algoritmi din teoria grafurilor. Mulți dintre acești algoritmi implică operații complexe pe seturi mari de date, care pot consuma timp și necesită resurse de calcul semnificative. O abordare pentru eficientizarea acestor algoritmi este utilizarea tehnicilor de calcul paralel, cum ar fi programarea în paralel pe GPU.

O altă problemă ce privește procesarea grafurilor de mari dimensiuni este parcurgerea acestora și implică vizitarea tuturor nodurilor unui graf. Există mai mulți algoritmi pentru parcurgerea de grafuri, incluzând aici căutarea în lățime (BFS - Breadth First Search) și căutarea în adâncime (DFS - Depth First Search). Acești algoritmi pot fi paralelizați folosind programarea GPU și CUDA pentru a obține accelerații semnificative față de implementările secvențiale. De exemplu, cercetătorii au dezvoltat algoritmi paraleli BFS și DFS care pot atinge accelerații de până la 100 de ori pe GPU-uri în comparație cu implementările clasice pe CPU (Klukovich și alții 2016) (Wang, Dong și Yuan 2013).

Problema partiționării grafurilor este o altă provocare pentru procesarea acestora. Aceasta implică împărțirea unui graf în subgrafuri mai mici care pot fi procesate în paralel. Partiționarea grafului este un pas critic în mulți algoritmi și poate avea un impact semnificativ asupra performanței generale a algoritmului. Cercetătorii au dezvoltat mai mulți algoritmi paraleli de partiționare a grafurilor, care pot fi executați pe GPU-uri folosind CUDA. Acești algoritmi pot obține accelerații semnificative față de implementările secvențiale și pot fi utilizați pentru a procesa eficient grafuri foarte mari (Goodarzi, Burtscher și Goswami 2016).

Evoluțiile recente în utilizarea programării GPU și a CUDA pentru clustering-ul de grafuri, care implică gruparea vârfurilor într-un graf pe baza proprietăților lor structurale. Gruparea grafurilor este o problemă fundamentală în multe aplicații, inclusiv analiza rețelelor sociale și bioinformatica. Cercetătorii au dezvoltat câțiva algoritmi paraleli de grupare a grafurilor care pot fi executați pe GPU-uri folosind CUDA. Acești algoritmi pot obține accelerații semnificative față de implementările secvențiale și pot fi utilizați pentru a procesa eficient grafuri foarte mari (Auer și Bisseling 2012).

Rezumând, programarea pe GPU folosind CUDA în teoria grafurilor oferă un set puternic de instrumente pentru procesarea rapidă și eficientă a grafurilor la scară largă. Evoluțiile recente în programarea GPU și CUDA le-au permis cercetătorilor să dezvolte algoritmi paraleli de procesare a grafurilor care pot obține accelerări semnificative față de implementările secvențiale. Pe măsură ce dimensiunea și complexitatea grafurilor continuă să crească, ne putem aștepta să vedem dezvoltarea și adoptarea continuă a programării GPU și a CUDA în domeniul teoriei grafurilor.

O altă problemă abordată în această lucrare este obținerea de hărți georeferențiate plecând de la valori măsurate în puncte discrete într-o anumită zonă geografică și folosind metode de interpolare bidimensională. Pentru obținerea de hărți de precizie, cu rezoluție ridicată, metodele de interpolare au fost accelerate folosind programarea pe GPU.

Pe scurt, în această carte se prezintă în **Capitolul 1** câteva aspecte generale legate de programarea pe GPU, scurt istoric, avantaje și limitări ale acestei tehnologii, aplicații ale programării pe GPU și câteva caracteristici generale ale programării folosind arhitectura CUDA. O parte dintre aceste aspecte și aplicații au fost publicate în lucrarea (Spridon, Advances in CUDA for computational physics 2023). **Capitolul 2** este dedicat rezultatelor personale în calitate de coautor în ceea ce privește găsirea unor algoritmi pentru generarea de rețele aleatoare și rezultatele obținute în urma paralelizării acestora pe GPU, folosind CUDA. **Capitolul 3** prezintă problema căutării drumului cu pierderea minimă într-un graf, problemă propusă și rezolvată în lucrarea (Deaconu, Spridon și Ciupala 2023). În **Capitolul 4** este prezentată o adaptare a algoritmului Ford-Fulkerson pentru rezolvarea problemei determinării fluxului maxim într-o rețea generalizată. În **Capitolul 5** sunt prezentate rezultatele legate de paralelizarea unor algoritmi de interpolare, rezultate publicate în (Spridon, Deaconu și Ciupala, ICCSA 2023) și (Ciupala, Deaconu și Spridon 2021). În ultima parte a acestei cărți sunt prezentate concluziile referitoare la rezultatele obținute.

Capitolul 1. Calcul de înaltă performanță pe plăci grafice

În acest capitol se 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 mari 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).

GPU-urile au apărut în anii 1990 pentru a accelera procesarea grafică în aplicațiile de jocuri și grafică computerizată (S3 Video Boards 1992). Acestea erau inițial specializate pentru randare grafică și nu ofereau suport pentru programare generală. Începând cu anii 2000, oamenii de știință și cercetătorii au început să exploreze utilizarea GPU-urilor pentru alte scopuri decât procesare grafică. Au apărut astfel primele eforturi de programare pe GPU, cum ar fi GPGPU (General-Purpose computing on Graphics Processing Units), care au permis programatorilor să utilizeze GPU-urile pentru programare generală (Krüger și Westermann 2003) (Bolz et al. 2003). În 2006, NVIDIA a lansat CUDA (Compute Unified Device Architecture), o platformă de calcul paralel dezvoltată special

pentru GPU-urile lor (Fung și Mann 2005). CUDA a introdus un model de programare care permitea programatorilor să scrie cod pentru GPU folosind limbajul de programare CUDA C/C++. Aceasta a deschis noi perspective pentru programarea pe GPU în domenii precum machine learning, simulări științifice și alte aplicații care implică calcul intensiv.

În 2008, Khronos Group a lansat OpenCL (Open Computing Language). Acesta este un standard deschis pentru programarea pe dispozitive heterogene, inclusiv GPU-uri. OpenCL permite programatorilor să scrie cod care rulează pe diverse platforme hardware, inclusiv GPU-uri de la diferiți producători pentru calcule generale care se efectuează în mod normal pe CPU, pentru a folosi capacitatea de calcul adițională de pe hardware-ul dedicat, atunci când acesta nu este folosit (Du et al. 2012).

De-a lungul timpului au apărut și alte platforme și biblioteci pentru programarea pe GPU, cum ar fi AMD ROCm și Intel oneAPI. GPU-urile au evoluat continuu în ceea ce privește puterea de calcul, memoria și caracteristicile tehnice. Altfel spus, programarea pe GPU a revoluționat domeniul calculului paralel și a deschis noi posibilități în rezolvarea problemelor complexe. Astăzi, GPU-urile moderne sunt capabile să gestioneze sarcini paralele complexe și să ofere performanță impresionantă în domenii variate.

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, așa cum se arată în **Figura 1.1**. 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).

"Moore's Law is dead." - Jensen Huang, 2013

"It's the end of Moore's Law as we know it." - John Hennessy, Oct 2018

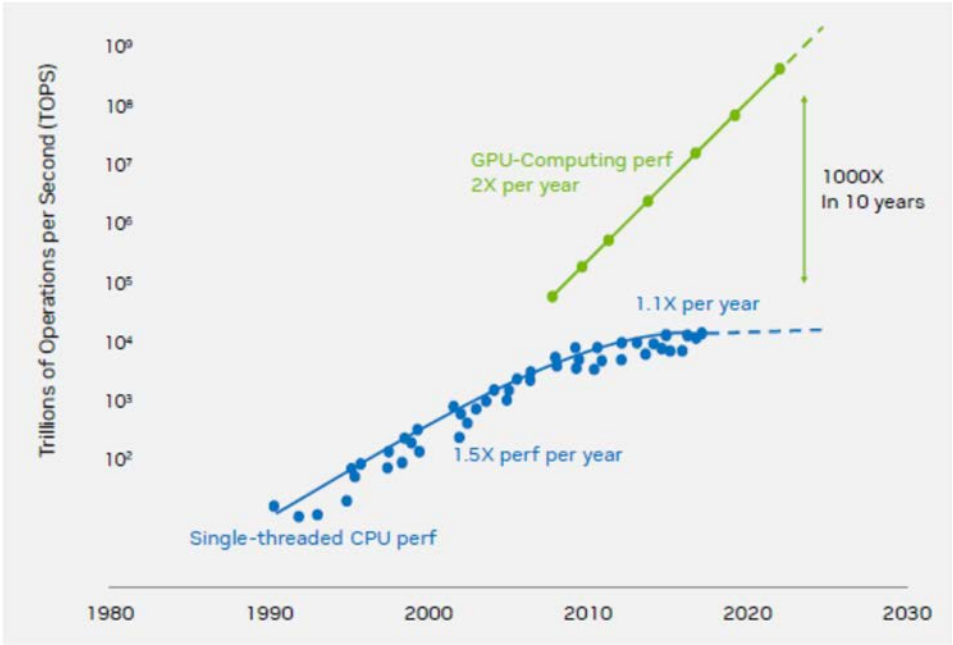


Figura 1.1 *Evoluția performanței GPU versus CPU* (Huang 2023)

Comparând cu unitățile centrale de procesare (CPU - Central Processing Unit), principalele avantaje ale programării pe GPU sunt:

- GPU-urile sunt concepute pentru a procesa simultan cantități mari de date, ceea ce le face mult mai rapide decât procesoarele pentru anumite sarcini, în special cele care implică seturi mari de date sau calcule complexe (Rathore et al. 2018) (Bharadiya 2023). Această procesare paralelă poate duce la o accelerare semnificativă pentru anumite tipuri de aplicații, cum ar fi procesarea imaginilor, învățarea automată și simulările științifice (Bali and Anandaraj 2023) (Bharadiya 2023) (Hancock, Khoshgoftaar and Johnson 2023).
- GPU-urile pot efectua un volum mare de calcule cu un consum mai mic de energie în comparație cu procesoarele. Acest lucru se datorează designului GPU-urilor, care le permit să gestioneze multe calcule simple în paralel și, astfel, GPU-urile să fie o opțiune mai eficientă din punct de vedere energetic pentru aplicațiile cu calcul intensiv (D'Agostino, et al. 2021).

- GPU-urile sunt relativ accesibile în comparație cu hardware-ul specializat (cum ar fi ASIC-urile sau FPGA-urile), ceea ce le fac o opțiune rentabilă pentru cercetători, dezvoltatori sau întreprinderile mici.
- Programarea GPU este posibilă în diferite limbaje, cum ar fi C/C++, Python și MATLAB, permițând dezvoltatorilor să aleagă limbajul și cadrul preferat (Weiss and Elsherbeni 2020).

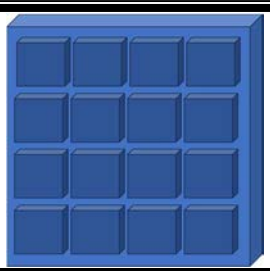
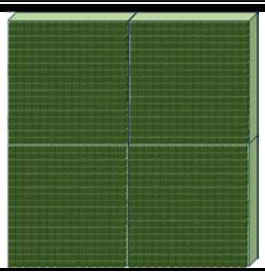
Există, de asemenea, și o serie de limitări ale programării pe GPU și anume:

- Programarea pe GPU necesită cunoștințe și experiență specializate, inclusiv înțelegerea arhitecturii GPU, managementul memoriei și tehnici de optimizare. Aceasta înseamnă că dezvoltatorii trebuie să investească timp și resurse în învățarea acestor abilități pentru a utiliza eficient GPU-urile pentru aplicațiile lor.
- GPU-urile sunt de obicei proiectate să funcționeze cu memoria lor dedicată, ceea ce necesită transferul frecvent de date între memoria RAM și memoria GPU. Acest lucru poate duce la o latență suplimentară, reducând viteza pe care o oferă GPU-urile în procesarea paralelă (Cugola și Margara 2012).
- Nu toate aplicațiile pot beneficia de programarea GPU. Aplicațiile care implică luarea deciziilor complexe sau operațiuni de ramificare, cum ar fi unele tipuri de sisteme de control sau software cu intrări imprevizibile, nu sunt potrivite pentru programarea pe GPU.
- Depanarea și determinarea profilului de rulare pe GPU pot fi mai dificile decât pe CPU (Kirk and Hwu 2016).
- Câștigurile de performanță ale programării GPU sunt limitate de dimensiunea GPU-ului și de capacitatea sa de memorie. Aplicațiile la scară largă pot necesita mai multe GPU-uri sau hardware specializat pentru a obține performanțe optime (Jog et al. 2015) (Sanders și Kandrot 2010).
- Diferitele arhitecturi GPU și API-uri necesită abordări specifice, ceea ce poate face dificilă portabilitatea codului între diferite platforme (Howes 2013).

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 dependente de limitările hardware, dar aplicațiile pe scară largă pot necesita hardware specializat sau mai multe GPU pentru a obține performanțe optime.

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 randării grafice, 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 de subliniat 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 (big data), simulări științifice, grafică și randare 3D, medicină și bioinformatică sau criptografie și securitate (**Figura 1.2**).

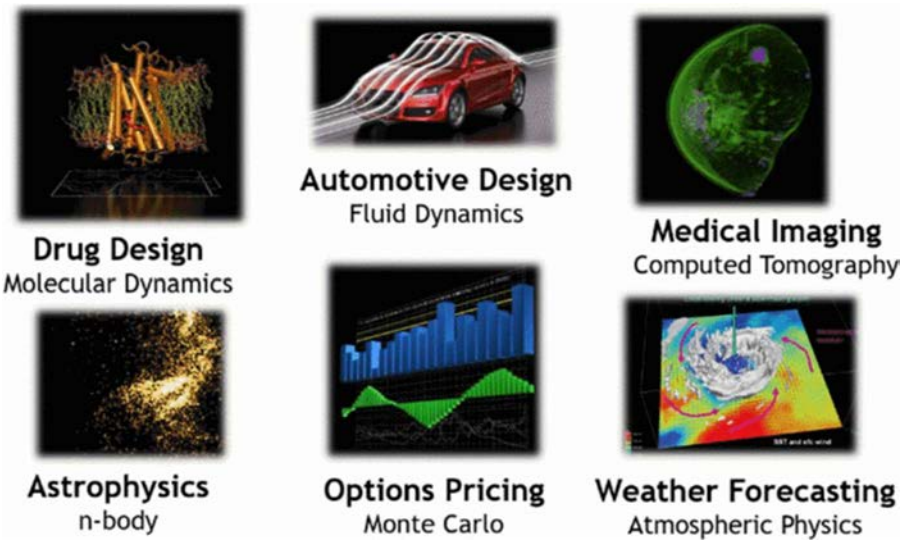


Figura 1.2 *Aplicații recente ale programării pe GPU* **(Baji 2018)**

În domeniul ML, GPU-urile sunt folosite intensiv în antrenarea rețelelor neuronale profunde și în realizarea de sarcini de învățare automată la scară largă (Gale, 2020). Framework-urile accelerate cu GPU, precum TensorFlow și PyTorch, sunt instrumente standard în cercetarea și dezvoltarea ML (Pang, Nijkamp și Wu 2020) (Prakash și Kanagachidambaresan 2021). Rețelele neuronale profunde (deep neural networks) sunt fundamentale în domeniul AI, dar antrenarea lor poate necesita resurse computaționale masive și timp considerabil (G. P. Zhang 2007). Cu ajutorul programării pe GPU, algoritmi precum propagarea inversă a erorii (backpropagation) și optimizarea gradientului pot beneficia de puterea de

calcul paralelă a GPU-urilor, accelerând procesul de antrenare a rețelelor neuronale (Liu și Guo 2013). După ce un model de AI este antrenat, el poate fi implementat și utilizat pentru inferență, adică pentru a face predicții și a lua decizii pe baza datelor de intrare (Prakash și Kanagachidambaresan 2021) (Pang, Nijkamp și Wu 2020) (G. P. Zhang 2007).

Programarea pe GPU permite rularea rapidă și eficientă a inferenței modelelor, permițând procesarea în timp real a datelor și implementarea soluțiilor AI în aplicații practice (Zhang and Li 2023). În plus, programarea pe GPU deschide calea pentru optimizarea și paralelizarea algoritmilor de ML, astfel încât să poată beneficia la maximum de puterea de calcul a GPU-urilor. Algoritmi precum k-means (Cuomo et al. 2019), random forest (Van Essen et al. 2012) și support vector machines (Athanasopoulos et al. 2011) pot fi implementați și rulați pe GPU pentru a obține performanțe mai bune și o scalabilitate mai mare. Pe scurt, programarea pe GPU este esențială în domeniul ML și AI pentru a obține performanțe ridicate și o scalabilitate eficientă. GPU-urile oferă o putere de calcul masivă și o paralelizare eficientă, facilitând antrenarea, evaluarea și implementarea modelelor de AI în aplicații practice.

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 de fizică computațională 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).

GPU-urile au fost proiectate inițial pentru randarea grafică, acestea fiind esențiale în industria jocurilor video și în animație, permițând proiectarea în timp real a graficii 3D complexe. Acestea sunt utilizate, de asemenea, în proiectarea asistată de calculator (CAD), sau vizualizarea datelor și grafică generată de computer (CGI).

Programarea pe GPU este folosită în aplicații criptografice, cum ar fi algoritmi de criptare, decriptare și hash, care necesită calcule intensive și paralele (Bharadiya 2023) (Wang și Chu 2019). De asemenea, GPU-urile sunt utilizate în simulări criptografice și în procesul de minare a criptomonedelor. Cercetări recente în acest sens propun utilizarea GPU-urilor pentru generarea rapidă a cheilor criptografice. Astfel, cercetătorii dezvoltă algoritmi și metode eficiente pentru generarea cheilor criptografice utilizând puterea de calcul paralel oferită de GPU-uri (Bharadwaj et al. 2021). Alte lucrări din acest domeniu investighează tehnici de optimizare și paralelizare pentru a accelera calculele de hash criptografic și a obține o performanță superioară în aplicațiile criptografice (Wang și Chu 2019).

În biologie, GPU-urile sunt folosite în analiza secvențelor genetice și în simulările moleculare pentru înțelegerea structurii și funcției proteinelor, dezvoltarea de medicamente și alte aplicații medicale (Zhang, Liu și Bu 2021) (Taylor-Weiner et al. 2019).

Alte domenii în care programarea pe GPU este utilizată în studii recente sunt:

- Domeniul financiar pentru analiza datelor financiare, cum ar fi prețurile acțiunilor, tranzacțiile și modelele de predicție a pieței. Aceasta permite

procesarea rapidă a datelor financiare și generarea de informații relevante pentru luarea deciziilor în timp real (Lee și Constantinides 2023) (Cai 2023).

- Imagistica medicală pentru analiza și procesarea imaginilor medicale, cum ar fi tomografia computerizată (CT), imagistica prin rezonanță magnetică (MRI) și imagistica moleculară (Guo et al. 2024) (Viola et al. 2023). Aceasta permite extragerea de informații importante din imagini medicale și asistența în diagnosticul și tratamentul bolilor.
- Realitatea virtuală și augmentată: procesarea grafică intensivă necesară pentru a oferi o experiență vizuală și interactivă captivantă (Fu et al. 2023) (Mandal et al. 2023).
- Analiza de rețea și securitate cibernetică: programarea pe GPU este folosită în analiza traficului de rețea (Barrionuevo, et al. 2015) (Feitoza Santos, et al. 2013) și detecția anomaliilor în scopul protejării rețelelor și a sistemelor informatice împotriva atacurilor cibernetice (Shevenell et al. 2014) (Lee et al. 2022).
- Rezolvarea problemelor matematice complexe, cum ar fi simularea fluxurilor de fluid (Wang, Abel și Kaehler 2010) (Bernaschi et al. 2010) sau modelarea prezicerii datelor meteorologice și climatice (Michalakes și Vachharajani 2008) (Vanderbauwhede și Takemi 2016).
- Procesarea semnalelor pentru comprimarea și decomprimarea video (Karpinsky și Zhang 2012) (Khani, Sivaraman and Alizadeh 2021), prelucrarea audio (Savioja, Välimäki și Smith 2011) (Belloch et al. 2014) și filtrarea semnalelor (Li, Bolic și Djuric 2015) (Nejedly et al. 2018).
- Analiza și vizualizarea datelor geospațiale, cum ar fi hărți, imagini de satelit și date topografice, pentru a extrage informații și a realiza analize complexe (Zhang, You și Gruenwald 2015) (Z. Li 2020).
- Modelarea 3D în procesul de design asistat de calculator permițând interacțiunea fluidă cu modelele complexe și redarea lor în timp real (Kersten și Stallmann 2012).

Acestea sunt doar câteva exemple de aplicații recente ale programării pe GPU, dar domeniul continuă să se extindă pe măsură ce se descoperă noi moduri de a exploata puterea de calcul paralelă a GPU-urilor în diverse industrii și discipline științifice.

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) este o platformă de calcul paralel dezvoltată de NVIDIA. CUDA oferă un mediu de programare care permite dezvoltatorilor să scrie cod C/C++ care rulează direct pe GPU. Folosind CUDA, programatorii pot utiliza puterea de calcul paralelă a GPU-urilor NVIDIA și pot accesa funcționalități avansate, cum ar fi memoria partajată, sincronizarea și comunicarea între blocurile de fire de execuție.
- OpenCL (Open Computing Language) este un standard deschis pentru programarea paralelă care poate fi utilizat pentru diverse dispozitive, inclusiv GPU-uri, CPU-uri și acceleratoare hardware. OpenCL oferă un model de programare flexibil, permițând dezvoltatorilor să scrie cod care poate fi rulat pe mai multe platforme și dispozitive diferite. Acesta este susținut de mai mulți producători de GPU-uri, inclusiv AMD, Intel și NVIDIA.
- OpenACC este un standard de programare pentru calculul paralel dezvoltat de Cray, CAPS, Nvidia și PGI. Acesta a fost conceput pentru a simplifica programarea paralelă a sistemelor CPU/GPU eterogene (O'Flaherty 2011). Abordarea de bază este de a insera comentarii speciale (directive) în cod, astfel încât să descarce calculul pe GPU-uri și să paralelizeze codul la nivelul nucleelor GPU (CUDA). Programatorii pot crea un cod OpenACC paralel eficient doar cu modificări minore la un cod CPU serial.
- SYCL (Single-source Heterogeneous Programming in C++) este un model de programare de nivel înalt care extinde C++ pentru a permite programarea paralelă pe diverse arhitecturi, inclusiv GPU-uri. SYCL utilizează un model de programare bazat fluxuri de date în rețele și oferă o abstractizare puternică a dispozitivelor hardware. Acesta permite dezvoltatorilor să scrie cod C++ portabil care poate rula pe diverse platforme și dispozitive.
- Vulkan este un API grafic și de calcul de înaltă performanță, dezvoltat de Khronos Group. Vulkan oferă o abordare modernă pentru programarea paralelă pe GPU, permițând accesul direct la funcționalitățile hardware ale GPU-urilor. Aceasta poate fi utilizată pentru calcul general, nu doar pentru grafică, și oferă control detaliat asupra resurselor și execuției paralele.

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 and Hamze 2010). Rezultatele lor sunt prezentate în **Figura 1.3**.

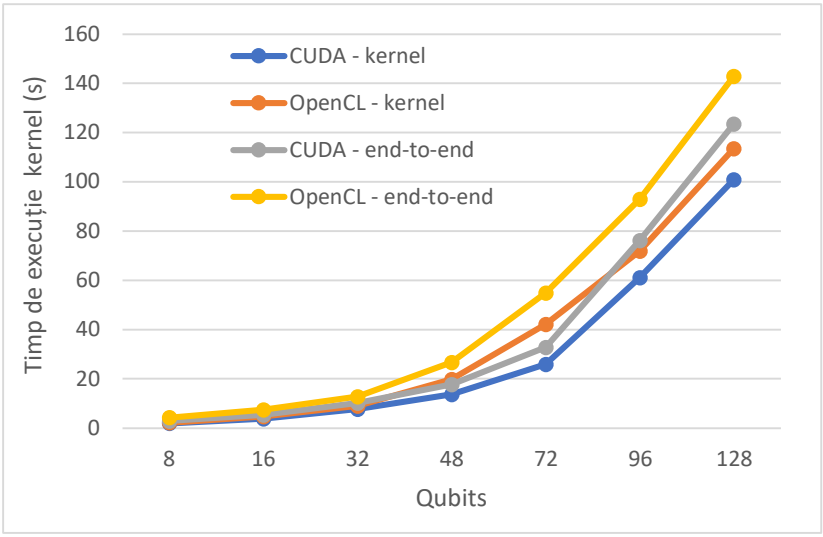


Figura 1.3 *Comparație timpi de execuție CUDA vs OpenCL*

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.

Hoshino et al. se concentrează pe aspectele de performanță ale OpenACC folosind două micro benchmark-uri și o aplicație de dinamică a fluidelor (Hoshino et al. 2013). Ambele evaluări arată că, în general, performanța OpenACC este cu aproximativ 50% mai mică decât CUDA. Cu toate acestea, pentru unele aplicații se poate ajunge la o viteza de execuție cu până la 98% mai mare dacă se fac anumite optimizări manuale atente în codul CUDA. Rezultatele indică, de asemenea, și

câteva limitări ale specificației OpenACC care împiedică utilizarea completă a resurselor hardware GPU, rezultând un decalaj semnificativ de performanță în comparație cu un cod CUDA complet reglat și optimizat. Lipsa unei interfețe de programare pentru memoria partajată are ca rezultat o performanță de trei ori mai mică conform autorilor acestei lucrări.

Khalilov și Timoveev au publicat o altă lucrare de studiu comparativ (Khalilov și Timoveev 2020). În aceasta se compară performanța CUDA, OpenMP și OpenACC pe GPU-ul Nvidia Tesla V100 în diferite scenarii tipice care apar în programarea științifică, cum ar fi înmulțirea matricilor, și se evaluează performanța codurilor de simulare implementate folosind aceste modele de programare. Rezultatele arată că, pentru dimensiuni mici ale grilei de calcul, timpii computaționali pentru CUDA și OpenACC sunt comparabili. Pe măsură ce domeniul de calcul crește, implementarea CUDA are rezultate mai bune. Astfel, la dimensiunea maximă a rețelei, între CUDA și OpenACC există o diferență de 40% în timpul de calcul în favoarea CUDA.

În 2023, Apanasevich et al. fac un studiu comparativ între CUDA și SYCL pentru Gromacs, un pachet software utilizat pe scară largă pentru simulări biomoleculare (Apanasevich et al. 2023). Analiza comparativă a performanței SYCL și CUDA pe Gromacs pentru diferite GPU-uri NVIDIA (V100, P100 și A100) folosind diferite seturi de date de apă cu proteine arată o performanță cel mult similară a SYCL față de CUDA. În 2021, Haseep et al. au folosit ADEPT, un kernel de aliniere cu citire accelerată de GPU, ca studiu de caz (Haseeb, et al. 2021). Ei au descoperit că implementarea SYCL rulează de aproximativ 2 ori mai lent decât omologul său CUDA în toate experimentele când se utilizează un GPU NVIDIA V100. Autorii atribuie această discrepanță capacității superioare de utilizare a memoriei cache de către CUDA și dependenței mai mari a SYCL de utilizarea regiștrilor.

Rezumând, 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 și 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 acest studiu a fost aleasă arhitectura

CUDA pentru performanța superioară anterior demonstrată în literatura de specialitate.

CUDA, după cum am menționat mai sus, este o platformă de calcul paralel dezvoltată de NVIDIA pentru a putea beneficia de puterea de calcul a GPU-urilor în aplicații de calcul general. Față de alte tehnologii enumerate mai sus, CUDA are avantajul că este optimizat pentru GPU-urile dezvoltatorului său, inclusiv seria NVIDIA GeForce, Quadro și Tesla. Astfel, CUDA are un ecosistem bogat și bine dezvoltat, inclusiv unelte, biblioteci și resurse comunitare extinse. Altfel spus, CUDA este o tehnologie care permite programatorilor să utilizeze GPU-urile NVIDIA pentru a accelera sarcinile de calcul intensiv, cum ar fi simulările științifice, analiza de date, machine learning și inteligența artificială. Această platformă oferă un model de programare flexibil, care permite programatorilor să scrie cod pentru GPU folosind limbajul de programare CUDA C/C++, un limbaj similar cu C/C++, dar care include și extensii specifice pentru a exploata paralelismul masiv oferit de GPU-uri.

CUDA permite programatorilor să utilizeze miile de nuclee de procesare ale GPU-urilor în mod eficient prin intermediul modelului de fire de execuție și blocuri și oferă mecanisme pentru gestionarea eficientă a transferului datelor între memoria RAM a sistemului și memoria globală a GPU-ului. Astfel, gestionând corespunzător nivelurile multiple de memorie de pe GPU, procesele pot fi eficientizate și accelerate.

Provocarea de a porta aplicațiile în CUDA este încă o problemă tehnică și practică de rezolvat pentru programatori. Într-o implementare CUDA a unui algoritm, programatorul trebuie să aibă grijă de gestionarea memoriei și a transferului de date între CPU și GPU, pentru a asigura o utilizare optimă a memoriei GPU și pentru a împacheta codul GPU în funcții separate. Un prim pas pentru crearea unui program CUDA este proiectarea unui algoritm care poate fi paralelizat. Aceasta implică împărțirea sarcinii în sub-sarcini mai mici care pot fi executate independent. Mai mult, un program bazat pe CUDA constă în două tipuri de coduri: setul de instrucțiuni care rulează pe CPU numit cod gazdă (*host* code) și setul de instrucțiuni care rulează pe GPU numit cod de dispozitiv (*device* code). Un workflow al unui protocol de lucru CUDA este prezentat în **Figura 1.4**. Aplicațiile încep să ruleze pe CPU și codul *host* gestionează și codul de tip *device*. Datele care trebuie procesate sunt încărcate în memoria GPU. Pentru aceasta, se alocă

memoria necesară pe GPU și datele sunt trasferate din memoria RAM în memoria GPU 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 blocuri de utilizat și numărul de fire de execuție pe fiecare bloc. 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 ne 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()". Odată ce kernel-ul și-a finalizat execuția, trebuie să transferăm datele înapoi de la GPU pe CPU la gazdă. În cele din urmă, memoria care a fost alocată pe GPU trebuie eliberată folosind „cudaFree()".

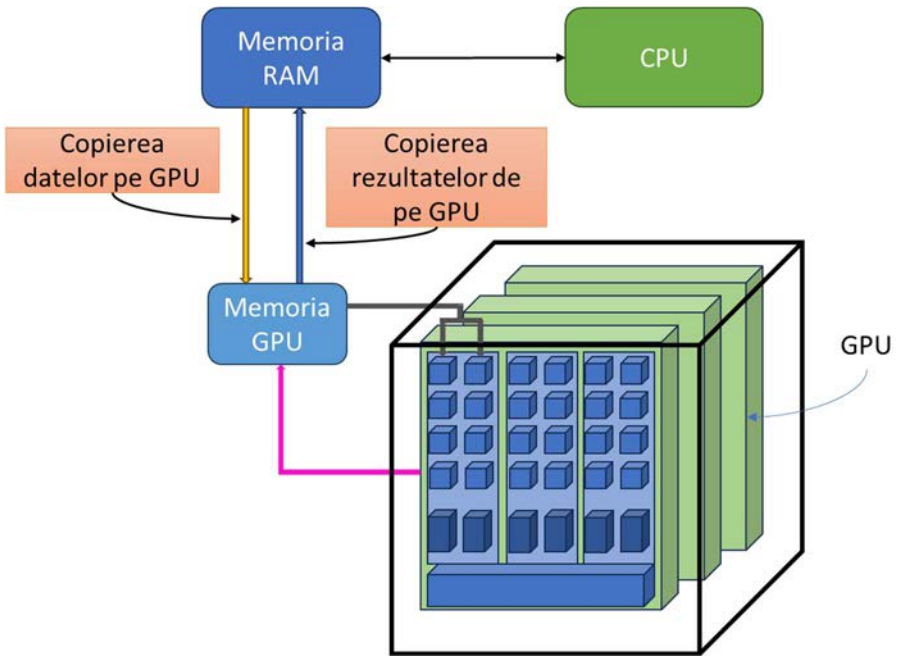


Figura 1.4 *Etapele de lucru pentru un program CUDA*

Pe scurt, datele sunt mai întâi trimise din memoria RAM în memoria GPU, apoi CPU-ul trimite instrucțiuni către GPU, GPU-ul programează și execută

kernel-ul pe hardware-ul paralel disponibil și, în sfârșit, rezultatele sunt copiate înapoi din memoria GPU-ului în memorie CPU (**Figura 1.4**).

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 1.5**.

- Memoria globală a GPU-urilor este memoria cea mai lentă de accesat, dar este cea mai mare cantitate disponibilă. Acesta poate fi gestionată de CPU folosind funcțiile cudaMalloc, cudaFree, cudaMemcpy sau cudaMemset și, prin urmare, este alocată / dealocată sau setată din CPU.
- Memoria constantă face parte din memoria principală a GPU-urilor și toate firele de execuție pot citi aceeași memorie constantă. Valorile memoriei constante sunt stabilite de CPU înainte de lansarea kernel-urilor.
- Memoria partajată poate fi accesată foarte rapid și este folosită pentru comunicații rapide între firele de execuție din interiorul unui bloc.
- Memoria locală este, de asemenea, lentă fiind parte a memoriei principale și este utilizată automat de firele de execuție atunci când memoria pentru regiștri nu mai este disponibilă.
- Regiștrii sunt variabilele declarate în nuclee și este memoria care se accesează cel mai rapid. Când se epuizează memoria pentru regiștri, se utilizează memoria locală.

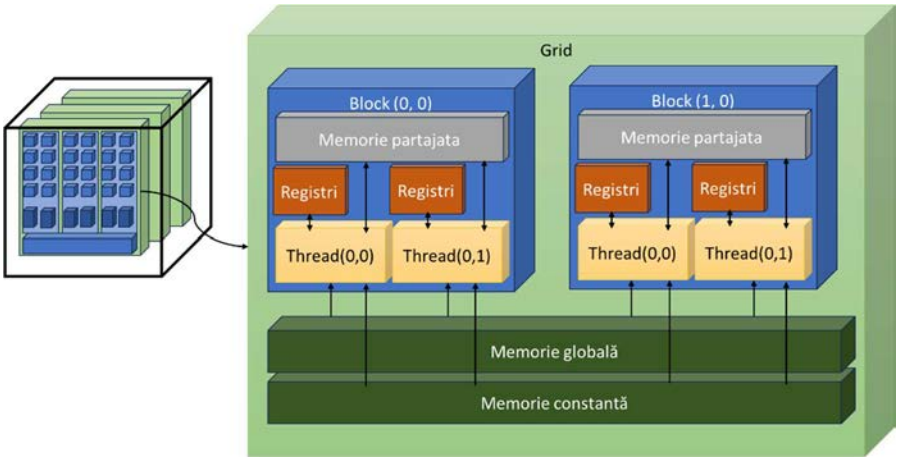


Figura 1.5 *Ierarhia memoriei GPU*

CUDA vine cu o serie de biblioteci și instrumente care facilitează programarea pe GPU. Printre acestea se numără cuDNN (CUDA Deep Neural Network library) pentru accelerarea rețelelor neuronale, cuBLAS (CUDA Basic Linear Algebra Subprograms) pentru operații algebrice de bază și cuFFT (CUDA Fast Fourier Transform) pentru transformate Fourier rapide.

CUDA suportă o gamă largă de GPU-uri NVIDIA, de la cele mai vechi modele până la cele mai recente. Acest lucru oferă programatorilor flexibilitate în alegerea hardware-ului potrivit pentru aplicațiile lor.

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 în 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 datorată utilizării programării CUDA.

2.1. Grafuri - generalități

2.1.1. Importanța grafurilor

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.

Grafurile sunt utilizate, de exemplu, pentru a modela rețelele sociale (Nettleton 2013), unde nodurile reprezintă utilizatori și muchiile reprezintă legăturile între utilizatori. Aceasta permite analiza proprietăților rețelei sociale,

cum ar fi influența socială, grupurile de interese comune, descoperirea comunităților. O altă aplicație a teoriei grafurilor este pentru modelarea rețelelor de transport (Ducruet și Lugo 2013), cum ar fi rețelele rutiere, rețelele de cale ferată sau rețelele aeriene pentru a găsi cel mai scurt drum între două puncte, pentru a planifica rutele optime, pentru a analiza fluxurile de trafic etc. Teoria grafurilor este utilizată și în proiectarea circuitelor electronice (i Cancho, Janssen și Solé 2001), unde nodurile reprezintă componente electronice, iar muchiile reprezintă conexiunile dintre acestea. Grafurile sunt folosite pentru a optimiza traseele circuitelor și pentru a verifica proprietățile circuitelor. Problemele de optimizare a rutelor, cum ar fi problema rutelor vehiculelor pot fi rezolvate, de asemenea, cu ajutorul algoritmilor din teoria grafurilor (Almasan et al. 2022). Aceasta implică găsirea celor mai scurte sau mai eficiente rute pentru a vizita un set de locații sau pentru a livra bunuri. Grafurile sunt utilizate și, pentru a analiza diverse tipuri de rețele, cum ar fi rețelele de comunicare (W. Jiang 2022), rețelele neuronale (Micheli 2009) sau rețelele metabolice (Wagner și Fell 2001). Aceasta permite înțelegerea structurii și comportamentului rețelelor, identificarea nodurilor critice sau a căilor importante și modelarea interacțiunilor între elemente.

Având în vedere cele de mai sus, putem spune că teoria grafurilor este o disciplină extrem de utilă și versatilă, cu aplicații într-o gamă largă de domenii, inclusiv în informatică, inginerie, logistică, biologie, fizică și multe altele.

Există multe aplicații ale problemelor legate de flux în rețele, cum ar fi cele electrice, de alimentare cu apă sau gaz, rutarea și transportul vehiculelor, rețelele wireless, extragerea de date, programarea companiilor aeriene, selecția proiectelor, segmentarea imaginilor, fiabilitatea rețelei, reconstrucția scenei cu mai multe camere, securitatea datelor statistice, predicția funcției genelor, calcul distribuit, conectivitate la rețea, detectarea intruziunilor în rețea, modele financiare, etc. (Mahmoudi și Boloori 2013) (Bingdong et al. 2013). Algoritmii pentru problemele de flux în rețea sunt dezvoltați și îmbunătățiți continuu. În consecință, este foarte important să existe un instrument de creare a rețelelor de testare a corectitudinii și de a compara timpii de execuție ai noilor algoritmi cu cei existenți.

2.1.2. Noțiuni teoretice

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, $\boldsymbol{G = (V, E)}$, 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 arc. Pentru un 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 un arc $a = (u, v)$ ale cărui noduri terminale sunt identice, adică $u = v$. Arcele multiple sunt un set de arce care au aceeași pereche de noduri terminale.

Definiție 2.2 Un **digraf** este un graf care nu are bucle sau muchii multiple.

Există mai multe criterii de clasificare a grafurilor. Astfel, după direcția de orientare a muchiilor există:

- *grafuri orientate* ale căror muchii sunt reprezentate din perechi ordonate de noduri, $E = \{(u, v) \in V \times V \; cu \; u \neq v\}$.
- *grafuri neorientate* ale căror muchii sunt reprezentate din perechi neordonate de noduri, $E = \{\{u, v\} \in \wp_2(V) \; cu \; u \neq v\}$.

După conectivitate, grafurile se clasifică în:

- *grafuri conexe* – în care există cel puțin un lanț între oricare două noduri ale grafului.
- *grafuri neconexe* – în care există cel puțin două noduri între care nu există niciun lanț.

Definiție 2.3 Se numește **lanț** de la nodul u_1 la nodul u_{k+1} o secvență de arce $L = (a_1, \ldots, a_k)$ cu proprietatea ca fiecare arc a_i este de forma (u_i, u_{i+1}) sau (u_{i+1}, u_i). Numărul de arce din secvență reprezintă lungimea lanțului. Un lanț se numește simplu dacă fiecare arc al lanțului, a_i, este utilizat o singura dată și se numește elementar dacă fiecare nod al lanțului, u_i, este utilizat o singura dată.

Definiție 2.4 Un **drum** sau **lanț orientat** este un lanț în care fiecare arc a_i este un arc direct (adică de forma (u_i, u_{i+1})).

Definiție 2.5 Un **ciclu** este un lanț simplu, L în care $u_1 = u_{k+1}$, adică cele două capete ale lanțului, coincid.

Definiție 2.6 Un lanț orientat, simplu, în care capetele coincid se numește **ciclu orientat** sau **circuit**.

2.1.3. Grafuri aleatoare

În literatura de specialitate sunt propuse câteva metode folosite pentru a construi *grafuri aleatoare*.

Definiție 2.7 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 oricărui arc 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. Un algoritm pentru generarea de grafuri aleatoare simple cu o anumită secvență de grade a fost dezvoltat în lucrarea publicată de Bayati et al. (Bayati, Kim și Saberi 2010). Folosind acest algoritm, se generează foarte rapid un graf aleatoriu uniform asimptotic cu o secvență de grade dată (timp aproape liniar). 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 $\boldsymbol{m_0}$ noduri date. Secvențial, nodurile sunt introduse în rețea. Fiecare dintre aceste noduri nou adăugate este conectat la $\boldsymbol{m} \leq \boldsymbol{m_0}$ noduri existente folosind o probabilitate dată, care este proporțională cu numărul de conexiuni pe care le aveau deja nodurile adăugate anterior. Probabilitatea $\boldsymbol{p_i}$ de a conecta un nou nod la nodul $\boldsymbol{i}$ este:

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

unde k_i este așa-numitul grad al nodului i. Numitorul din ecuația (2.1) este de două ori numărul de arce existente în rețea.

Penrose, M. a introdus un așa-numit graf geometric aleatoriu (RGG – Random Geometric Graph), care este un graf geometric neorientat cu noduri eșantionate aleatoriu. Pentru a genera un astfel de graf, se folosește o distribuție uniformă a spațiului subiacent $[0, 1)^d$, unde d este dimensiunea spațiului (Penrose 2003). Ideea din spatele generării unui RGG este că două noduri sunt legate numai dacă distanța dintre ele este mai mică decât un parametru dat $r \in (0, 1)$. Prin urmare, r și n dau modul în care este generat un RGG. Foarte recent, RGG-urile au fost aplicate cu succes în nanomateriale (Aguilar-Sánchez et al. 2020). Waxman a generalizat RGG-urile luând în considerare o funcție de conexiune probabilistică (Waxman 1988).

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. Problema fluxului maxim este de a găsi un flux de la nodul sursă la nodul stoc având valoarea maximă posibilă. Foarte recent, au fost dezvoltați algoritmi din ce în ce mai buni pentru a rezolva această problemă (Gao, Liu și Peng 2021), (Brand et al. 2020), (Orlin 2013).

Împreună cu fluxul maxim, se poate calcula și tăietura minimă (Ahuja, Magnanti și Orlin 1993). Problema fluxului de cost minim este de a găsi un flux cu cost minim de la nodurile sursă la nodurile stoc. Recent, a fost dezvoltat cel mai bun algoritm pentru rezolvarea acestei probleme (Yongwen et al. 2020). Problema fluxului cu mai multe mărfuri utilizează cereri de flux sau mai multe mărfuri între diferite noduri sursă și noduri receptor. Cel mai cunoscut algoritm pentru rezolvarea acestei probleme este de la Karakostas (Karakostas 2002). Există și alte probleme de flux ale căror algoritmi pot fi încă îmbunătățiți, de exemplu, problemele inverse de flux maxim generalizat (Tayyebi și Deaconu 2019).

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 - Graphic Neural Networks) - un tip de algoritm de învățare automată care poate fi folosit pentru a rezolva probleme legate de grafuri. Cercetătorii au explorat utilizarea programării GPU pentru implementarea GNN-urilor, rezultând o accelerare și o scalabilitate semnificative în comparație cu abordările tradiționale bazate pe CPU (Zonghan Wu, 2021) (Tianfeng Liu, 2023).
- Partiționarea grafurilor - o problemă fundamentală în teoria grafurilor care implică împărțirea unui graf în mai multe subgrafuri. Au fost dezvoltați algoritmi accelerați de GPU pentru partiționare care pot gestiona grafuri la scară mare cu milioane de vârfuri și arce (Santosh Nage, 2015).
- Numărarea triunghiurilor dintr-un graf - o problemă comună în teoria grafurilor. Cercetătorii au dezvoltat algoritmi accelerați de GPU pentru numărarea triunghiurilor care pot obține o accelerare de câteva ordine de mărime în comparație cu abordările bazate pe CPU (Liu Hu, 2021).
- Algoritmi pentru găsirea drumului cel mai scurt între două noduri dintr-un graf. Astfel de algoritmi au fost accelerați de GPU pentru grafuri la scară mare

și se pot obține o accelerări semnificative în comparație cu abordările tradiționale bazate pe CPU (Carl Yang, 2022).

- Algoritmul PageRank - un algoritm popular pentru clasificarea paginilor web în funcție de importanța lor. În (Seunghwa Kang, 2022) acest algoritm a fost implementat pe GPU pentru gestionarea grafurilor cu un număr mare de noduri și muchii și s-a obținut o accelerare semnificativă în comparație cu abordările tradiționale bazate pe CPU.

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

2.3.1. Descompunerea fluxului în fluxuri elementare

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.

Un flux (admisibil) într-o rețea s-t orientată G este o funcție $f : E \to \mathbb{R}_+$ care satisface condițiile de mărginire (2.2) și condițiile de conservare (2.3).

$$0 \leq f(u, v) \leq c(u, v) \; \forall (u, v) \in E \tag{2.2}$$

$$\sum_{v\in V,(u,v)\in E} f(u,v) - \sum_{v\in V,(v,u)\in E} f(v,u) = \begin{cases} 0, if u, v \in V - \{s,t\} \\ v_f, if\ u = s \\ -v_f, if\ v = t \end{cases} \quad (2.3)$$

unde $v_f = v(f) \geq 0$ este valoarea fluxului f.

Un flux admisibil poate fi descompus în două fluxuri admisibile f_1 și f_2 și este notat cu $f = f_1 + f_2$, dacă $f(a) = f_1(a) + f_2(a), \forall\, a \in E$.

$P = (u_1, u_2, \dots, u_k)$ se numește *drum* în G dacă $(u_i, u_{i+1}) \in E\ \forall\ i \in \{1,2, \dots, k-1\}$, unde $k \geq 2$. Un drum se numește *elementar* dacă nu trece de două ori prin același nod, adică $u_i \neq u_j\ \forall\ i, j \in \{1,2 \dots., k\}, i \neq j$. Un *circuit* este un drum pentru care nodul de intrare este același cu nodul stoc, adică, $C = (u_1, u_2, \dots, u_k = u_1)$ este un circuit. Un circuit este *elementar* dacă nu trece de două ori prin același nod, cu excepția nodului de intrare. Un flux este *elementar* dacă este 0 pe toate arcele rețelei, cu excepția arcelor de pe un drum $s - t$ sau a unui circuit, unde este egal cu o valoare $v > 0$.

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).

Pentru a arăta ideea din spatele **Teoremei 2.1**, în **Figura 2.1** este prezentat un flux f în rețeaua G. Fluxul este admisibil deoarece satisface ambele condiții (2.2) si (2.3). Valoarea fluxului $v_f = 6$. O posibilă descompunere a fluxului f în fluxuri elementare este dată de fluxurile f_1, f_2 și f_3 corespunzătoare respectiv drumurilor $P_1 = (1,2,4,6)$, $P_2 = (1,3,5,6)$ și circuitului $C = (3,5,4,3)$. Valoarea lui f_1 este 2 și este egală cu valoare pe drumul P_1 . Valoarea lui f_2 este 4 și este egală cu valoare pe drumul P_2 . Valoarea lui f_3 este 0, dar valoarea pe circuitul C este egală cu 2.

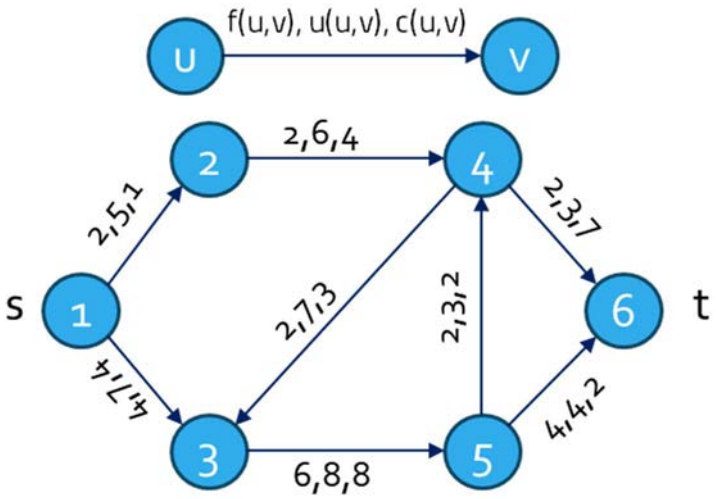


Figura 2.1 *Exemplu de rețea, G, și flux, f*

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.

2.3.2. Algoritm pentru generarea de rețele s-t aleatoare de flux

În cele ce urmează se prezintă algoritmi pentru generarea de drumuri și circuite elementare într-o rețea (Deaconu și **Spridon** 2021). 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 $v \in V$ **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 corespunzător 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 $j \neq u0$ **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 arcul (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 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 și **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 $u \neq v$ **atunci**
        $swap = nodes[u];$
        $nodes[u] = nodes[v];$
        $nodes[v] = swap;$
    sfârșit dacă;
sfârșit pentru;

AAVN începe cu un vector având toate nodurile cu indici de la i_{start} la i_{end}. Apoi, acest vector este amestecat alegând succesiv în mod aleatoriu două noduri din vector și interschimbându-le pozițiile acestora. Aceste interschimbări sunt

executate de n ori, unde $n = i_{end} - i_{start} + 1$ este lungimea vectorului de amestecat.

Avem următoarea teoremă care demonstrează calitatea vectorului amestecat obținut:

Teorema 2.2 Folosind algoritmul AAVN, orice vector de noduri generat aleatoriu are aceeași probabilitate de a fi generat.

Demonstrația Teoremei 2.2. Să presupunem că avem n valori care trebuie generate folosind AAVN. Vectorul inițial al nodurilor conține n valori distincte. Există n operații de schimbare aleatoare aplicate vectorului. Vom demonstra că orice permutare a valorilor inițiale poate fi obținută astfel.

Fie $p = (p_1, p_2, \dots, p_n)$ o permutare a valorilor inițiale $i_{start}, i_{start} + 1, \dots, i_{end}$.

Următorul algoritm (**Algoritmul 2.4**) transformă nodurile vectorului inițial în p.

Algoritmul 2.4 *Algoritm pentru permutarea nodurilor (APN)*

```
/* generarea permutării de noduri, p_k */
Pentru k = 1,n execută
    Găsește indexul i în nodurile permutării p_k;
    swap = nodes[i];
    nodes[i] = nodes[k];
    nodes[k] = swap;
Sfârșit pentru;
```

Folosind APN, există n operații de inter-schimbare care transformă nodurile în p. AAVN efectuează n operațiuni de permutări aleatoare la noduri. Deci, există întotdeauna o șansă ca AAVN să genereze p din noduri. Probabilitatea de a genera p din noduri este $1/n!$ folosind AAVN și, deoarece numărul total de permutări posibile este $n!$, rezultă că orice permutare a nodurilor vectoriale are șanse egale de a fi generată folosind AAVN.

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

Algoritmul 2.5 *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 sursă 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;
$ma[nodes[l_{path} - 2]][t] = 1$;

În **Algoritmul 2.5** (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 $2 \leq l_{path} \leq n$. 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 2.6 *Algoritm de generare a unui circuit aleatoriu – v2 (AGCA2)*

/ generare eficientă a unui vector de noduri */*
$AAVN(0, n-1)$;
/ generarea aleatorie a lungimii circuitului */*
$l_{cycle} = random(2, n)$;
/ se adaugă arcele care se formează între primele l_{cycle} noduri din vectorul de noduri amestecate ale rețelei */*

Pentru $k = 0, l_{cycle} - 2$ **execută**

$ma[nodes[k]][nodes[k + 1]] = 1;$

sfârșit pentru;

$ma[nodes[l_{cycle} - 1]][nodes[0]] = 1;$

În **Algoritmul 2.6**, 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 2.7** de generare a unei rețele de flux aleatorie.

Algoritmul 2.7 *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_{cycle} **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 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 Erdõs–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;

Înainte de a porni execuția AGRFA, matricea de adiacență ma este setată la valoarea 0. Algoritmul construiește ma și apoi listele de adiacență la, folosind ma.

După ce construirea drumurilor $s - t$ și a circuitelor, sunt adăugate aleatoriu arcuri în rețea folosind modelul Erdõs–Rényi. Conform acestui model, probabilitatea de a adăuga un nou arc este $p \in [0, 1)$. În consecință, în AGRFA, pentru fiecare pereche de noduri $(i, j), i \neq j$, astfel încât $ma[i, j] = 0$, adică (i, j) nu este în prezent un arc în rețea, un număr întreg este generat în intervalul $[0, 999]$ folosind o funcție de random, iar dacă această valoare este mai mică decât $p \cdot 1000$, arcul (i, j) este adăugat la rețea.

Capacitățile arcelor sunt generate aleatoriu între valorile min_c și max_c. Costurile pentru arce sunt, de asemenea, generate aleator între min_w și max_w. Există mai mulți parametri pentru unele probleme de flux, cum ar fi cele de tăietură minimă (Ahuja, Magnanti și Orlin 1993), (Deaconu și Ciupala, Inverse Minimum Cut Problem with Lower and Upper Bounds. 2020), problema inversă pentru flux maxim (Deaconu și Ciurea, The inverse maximum flow problem under Lk norms 2012), (A. Deaconu, A Cardinality Inverse Maximum Flow Problem 2006), rezistența arcelor (Marinescu, Deaconu et al., From microgrids to smart grids: Modeling and simulating using graphs.Part I active power flow 2010), (Marinescu, Deaconu et al., From Microgrids to Smart Grids: Modeling and Simulating using Graphs. Part II Optimization of Reactive Power Flow 2010) sau factorul de câștig (Durstenfeld 1964), (Sven și Zeck 2013). Aceste valori pot fi, de asemenea, generate aleatoriu pe arce.

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

Demonstrația teoremei 3. Complexitatea generării unui drum $s - t$ sau a unui circuit folosind AGDA2, respectiv, AGCA2 este $O(n)$. În consecință, matricea de adiacență *ma* este generată în $O(max\{n_{path}, n_{cycle}\} \cdot n)$, și, deoarece generarea listelor de adiacență ia $O(n^2)$ timp, complexitatea temporală a algoritmului 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, complexitatea este cel mai probabil $O(n^2)$.

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 nuclee de 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.

Rețeaua din **Figura 2.2** a fost generată folosind AGRFN. Parametrii de intrare au fost următorii: $p\ =\ 0.2$, $n_{path} = 4$, $n_{cycle} = 2$, $min_c = 1$, $max_c = 7$, $min_w = 1$ și $max_w = 7$. Algoritmul a generat următoarele drumuri: $P1 = (1, 2, 3, 6)$, $P2 = (1, 6)$, $P3 = (1, 4, 5, 6)$ și $P4 = (1, 2, 4, 6)$, precum și circuitele $C1 = (4, 5, 2, 3, 4)$ și $C2 = (2, 4, 5, 1, 2)$. În cele din urmă, din restul de 19 perechi de noduri care nu au fost conectate cu arce, pe baza probabilității considerate de $p\ =\ 0.2$, AGRFA a generat încă trei arce: $(1, 5), (3, 4)\ și\ (3, 5)$.

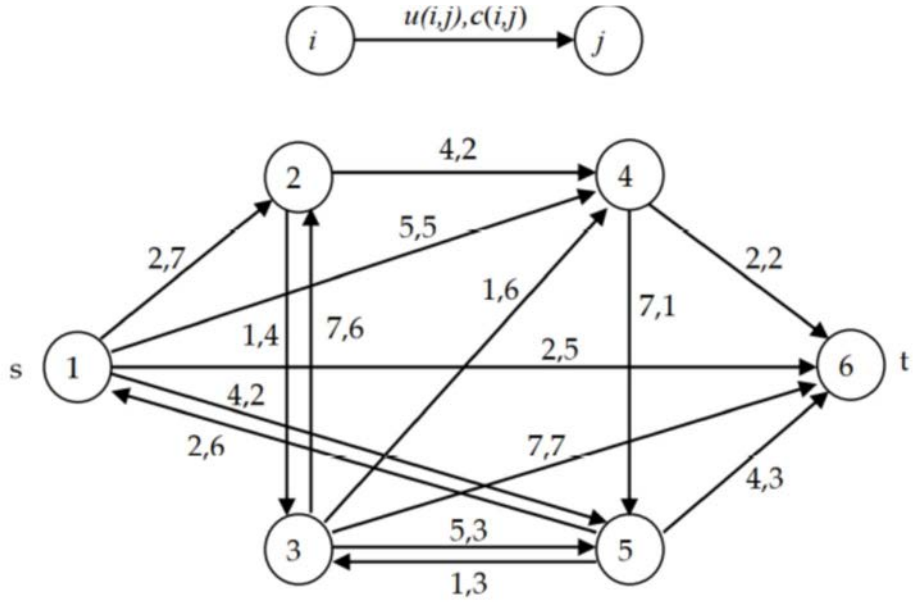


Figura 2.2 *Rețea generată cu AGRFA* (Deaconu și **Spridon** 2021)

2.4. Rezultate și discuții

În **Figura 2.3,** 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. După cum era de așteptat și cum arată **Tabel 2.1**, numărul de noduri împreună cu numărul de drumuri și circuite considerate influențează direct viteza de generare a rețelei. Pentru experimente 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 de execuție 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. Cea mai bună accelerare a fost obținută pentru rețelele mari, având mii de noduri (**Tabel 2.1).**

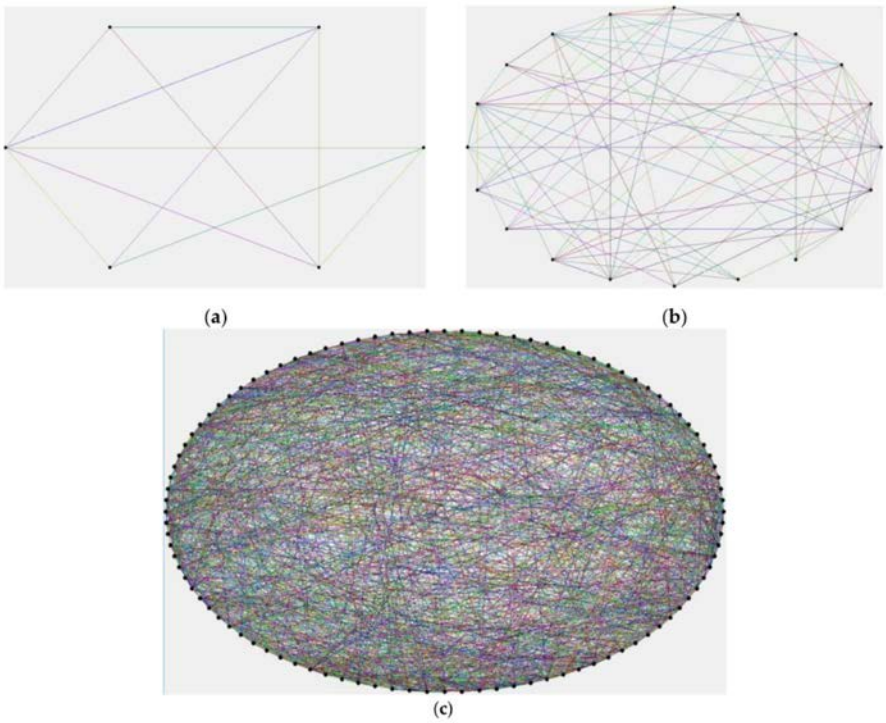


Figura 2.3 *Rețele generate folosind AGFFA. (a) Rețea cu n = 6, n_{path} = 3, n_{cycle} = 2; (b) Rețea cu n = 20, n_{path} = 10, n_{cycle} = 2; (c) Rețea cu n = 100, n_{path} = 20, n_{cycle} = 10.* (Deaconu și **Spridon** 2021)

Tabel 2.1 *Comparație timpilor de execuție și speed-up-ul pentru generarea rețelelor aleatoare*

Nr. de rețele generate	Nr. de nod-uri	Nr. de dru-muri	Nr. de circuite	Timp CPU (s)	Timp CUDA (s)	Nr. de block -uri	Thread-uri per block	Speed-up
1,000,000	10	5	2	8.42	69.50	1	10	0.12
100,000	50	20	10	14.39	10.99	1	50	1.31
50,000	100	50	25	31.33	8.64	1	100	3.62
4,000	500	200	100	44.66	6.10	1	500	7.32
1,000	1,000	500	250	55.33	4.09	1	1000	13.54
50	5,000	2000	1,000	58.19	3.20	5	1000	18.16
10	10,000	5000	2,500	57.33	3.01	10	1000	19.05

În **Figura 2.4** 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.

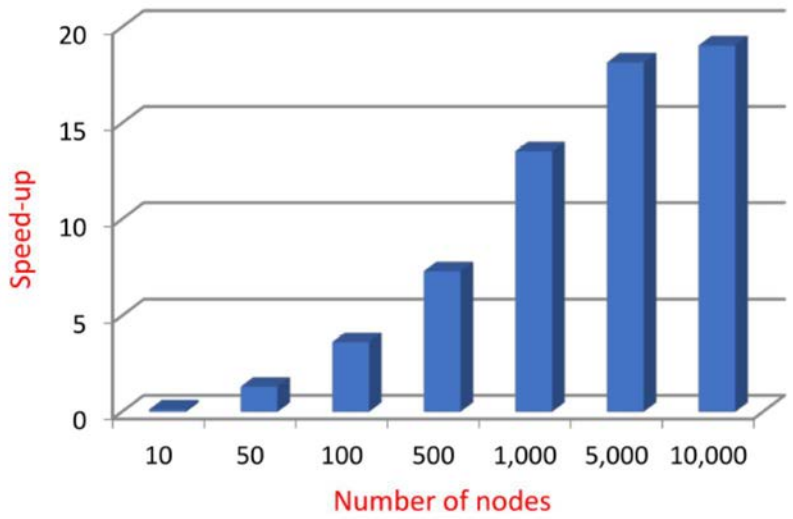


Figura 2.4 *Speed-up CPU/ CUDA* (Deaconu și **Spridon** 2021)

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 este propus un algoritm pentru determinarea drumului cu pierderi minime într-o rețea în care, pe fiecare arc pot exista scurgeri sau pierderi. Este studiat, de asemenea, și cazul în care pe arcele rețelei pot exista câștiguri / infuzii de flux.

3.1. Context științific

În viața de zi cu zi se întâlnesc tipuri variate de rețele, cum ar fi rețele telefonice, electrice, autostrăzi, rețele feroviare, de computere, de apa sau canalizare, etc. În toate aceste rețele, în general se dorește a se trimite eficient o cantitate energie, marfă, apă, etc (în funcție de rețeaua în care are loc transportul) pe care o numim în mod generic flux, de la un nod la altul, supunându-se totodată anumitor constrângeri. Pentru rezolvarea unor astfel de probleme de-a lungul timpului au fost proiectați o serie de algoritmi eficienți din punct de vedere computațional.

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 două 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 generalizată pentru fluxul maxim poate fi definită astfel:

Fie o rețea orientată $G = (V, E)$, unde V reprezintă mulțimea nodurilor și E reprezintă mulțimea arcelor, fiecare arc având capacitatea $c(u, v)$ și costul sau o altă măsură asociată cu fluxul, $w(u, v)$. Problema generalizată a fluxului maxim implică găsirea unui flux, $f(u, v)$, pentru fiecare arc (u, v) astfel încât să fie îndeplinite următoarele condiții:

(a) Capacitatea arcului: $0 \leq f(u, v) \leq c(u, v)$ pentru orice $(u, v) \in E$.

(b) Conservarea fluxului: pentru orice nod, u, diferit de sursă și stoc, suma fluxurilor care intră în nod este egală cu suma fluxurilor care ies din nod.

Scopul este de a maximiza funcția obiectiv care, pe lângă capacitatea fiecărui arc, poate implica și costuri, pierderi sau alte mărimi care pot fi asociate cu fluxul.

Pentru rezolvarea acestei probleme pot fi utilizați algoritmi specializați pentru problemele de flux maxim, precum algoritmul Ford-Fulkerson sau algoritmul Edmonds-Karp, dar adaptați în așa fel încât să ia în considerare și celelalte caracteristici ale rețelei.

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 cercetările efectuate de autoare, aceste două probleme sunt singurele probleme de optimizare care au fost analizate în rețele cu pierderi sau câștiguri pe arce.

În acest capitol, se introduce și se rezolvă 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 $s - t$.

3.2. Problema găsirii drumului cu pierdere minimă

3.2.1. Problema găsirii drumului cu pierdere minimă într-o rețea cu pierderi

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 A 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 **Figura 3.1** este prezentat un exemplu în care în nodul i ajung 80 de unități de flux, pe arcul (i, j) se pierd 0.8 unități. Astfel, în nodul j ajung doar 72 de unități. Pe arcul (j, k) se pierd încă 3.6 unități, nodul k rămânând astfel cu 68.4 unități.

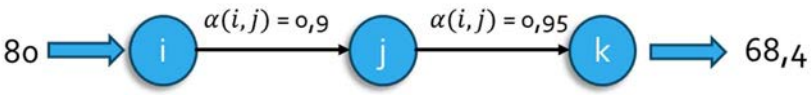


Figura 3.1 *Exemplu de rețea în care există pierderi pe arce*

Notez cu Π mulțimea de drumuri de la s la t, astfel:

$$\Pi = \{P = (v_0 = s, v_1, \dots, v_k) | \, v_i \in V (i = 0,1,2, \dots, k),$$

$$(v_i, v_{i+1}) \in E(i = 0, 1, 2, \dots, k-1)\} \qquad (3.1)$$

În cele ce urmează, se va prezenta o metodă de calculare a drumului cu pierdere minimă (MLP - Minimim Loss Path) sau drumului cu rata de livrare maximă (MDRP - Maximum Delivery Rate Math) de la nodul sursă, s, la nodul stoc, t. Notăm această problemă ca MLPP (Minimul Loss Path Problem). Pentru a rezolva MLPP, trebuie să se identifice 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 găsită soluția următoarei probleme de optimizare:

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

Pentru rezolvarea problemei de mai sus se transformă problema (3.2) într-o problemă de minimizare, astfel:

$$\begin{cases} min\{-\log\left(\alpha(v_0, v_1) \cdot \ \alpha(v_1, v_2) \cdot \dots \cdot \alpha(v_{k-1}, v_k)\right)\} \\ P = \ (v_0 = s, v_1, \dots, v_k) \in \Pi \end{cases} \qquad (3.3)$$

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

Problema anterioară poate fi rescrisă sub forma:

$$\begin{cases} min\{\beta(v_0, v_1) + \ \beta(v_1, v_2) + \dots + \beta(v_{k-1}, v_k)\} \\ P = \ (v_0 = s, v_1, \dots, v_k) \in \Pi \end{cases} \qquad (3.4)$$

unde:

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

Având în vedere faptul că valorile β atașate arcurilor sunt pozitive, este ușor de observat acum că problema (3.3) 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 $G = (V, E)$
- $\alpha(i, j) \in (0,1], (i, j) \in E$

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 $G = (V, A, \beta)$

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.

În **Algoritmul 3.1** se calculează inițial funcția β pentru fiecare arc al rețelei. De asemenea, se introduce, a priori, un test de fezabilitate pentru MLPP. Acest test poate fi efectuat în timp $O(m + n)$ folosind un algoritm de parcurgere în graf (Ahuja, Magnanti și Orlin 1993). În cazul în care nodul stoc este accesibil din nodul sursă, se aplică un algoritm pentru determinarea celui mai scurt drum în rețeaua nou formată $G = (V, E, \beta)$. Drumul găsit este și drumul cu pierderi minime în rețeaua inițială.

3.2.2. Problema găsirii drumului cu pierdere minimă într-o rețea generalizată

Se investighează, 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 se seteze $\alpha(u,v) > (0,1]$, se poate 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 .

În **Figura 3.2** este prezentat un exemplu de rețea în care există câștig pe arce. După cum se poate observa, în nodul i ajung 80 de unități de flux, pe arcul (i,j) se câștigă 0.8 unități. Astfel, în nodul j ajung 88 de unități. Pe arcul (j,k) se adaugă încă 3.6 unități, în nodul k ajungând 92.4 unități.

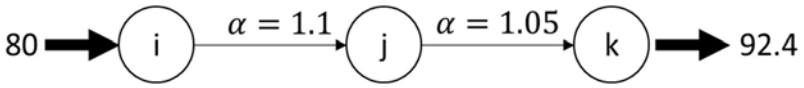


Figura 3.2 *Exemplu de rețea în care există câștig de flux pe arce*

Această problemă de optimizare are același model matematic (3.2) și poate fi, de asemenea, transformată în problema de minimizare (3.4). 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 ciclu 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

/ 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 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.

3.3.1. Algoritmul lui Dijkstra

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 începe de la un nod sursă și construiește treptat un arbore cu cele mai scurte drumuri de la sursă la toate celelalte vârfuri din graf. La fiecare pas, algoritmul selectează nodul nesetat cu cea mai mică distanță de la sursă și apoi examinează nodurile adiacente pentru a-și actualiza distanțele în cazul în care se găsește un drum mai scurt prin vârful curent. Algoritmul continuă până când toate nodurile sunt setate sau până când nodul stoc este analizat (setat).

Algoritmul lui Dijkstra nu poate fi aplicat pentru rețele cu valori negative ale arcelor; este un algoritm de tip greedy și garantează găsirea drumului minim într-un graf cu valori pozitive ale costurilor arcelor. Dacă inițial, algoritmul a fost utilizat pentru probleme de rutare în rețele de telecomunicații și transport, ulterior, algoritmul lui Dijkstra a influențat dezvoltarea altor algoritmi de rutare și optimizare, cum ar fi algoritmul A* și diverse tehnici de programare dinamică. Complexitatea sa a fost îmbunătățită de-a lungul anilor prin utilizarea structurilor de date mai eficiente, cum ar fi heap-urile binare sau Fibonacci, reducând astfel timpul de execuție pentru grafuri mari.

3.3.2. Algoritmul Bellman-Ford

Pentru a determina drumul cel mai scurt într-un graf care conține arce cu valori negative ale costurilor se poate aplica algoritmul **Bellman-Ford** (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.

Algoritmul Bellman-Ford începe prin inițializarea distanțelor de la nodul sursă la toate celelalte vârfuri din graf cu infinit, cu excepția distanței de la nodul sursă la el însuși, care este setată la 0. Apoi, relaxează în mod repetat toate arcele din graf. Relaxarea înseamnă a verifica dacă există vreun alt drum mai scurt de la nodul sursă la un nod dat. Dacă se găsește un drum mai scurt, atunci distanța până la vârful dat este actualizată. Algoritmul repetă procesul de relaxare de n-1

ori, unde n este numărul de noduri din graf. Motivul pentru aceasta este că orice drum mai scurt din graf poate avea cel mult n-1 muchii. După n-1-a iterație, dacă există vreun nod care poate fi încă relaxat, atunci graful conține un circuit negativ. Un circuit negativ este un circuit de arce ale căror cost total este negativ. Dacă algoritmul găsește un circuit negativ, atunci drumul cel mai scurt între sursă și oricare alt nod din circuit nu poate fi găsit deoarece circuitul respectiv poate fi parcurs de infinit de multe ori, la fiecare parcurgere lungimea drumului fiind micșorată.

3.3.3. Paralelizarea algoritmilor pentru găsirea drumului minim

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, se 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-Arranzet et al 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.

Astfel, în implementarea paralelă a algoritmului Dijkstra, pentru fiecare nod sunt determinate arcele incidente către exterior de cost minim, într-o fază de precalculare. Algoritmul paralelizat are 4 pași.

1. pasul de inițializare;
2. pasul de relaxare - se apelează funcția kernel care reduce distanțele pentru nodurile rămase nesetate până la iterația curentă prin analizarea arcelor incidente către exterior ale nodurilor de analizat. Astfel, în această etapă, un fir de execuție GPU este atribuit fiecărui nod din graf, iar cele alocate nodurilor de analizat parcurg arcele incidente către exterior și reduc distanțele pentru nodurile din mulțimea de succesori, care nu au fost încă setate.
3. pasul de minimizare - kernel-ul determină cea mai mică distanță pentru fiecare nod comparând cu noile valori atribuite.
4. pasul de actualizare - kernel-ul stabilește noduri ale căror distanțe provizorii sunt mai mici sau egale cu valorile calculate anterior. Acest lucru se realizează prin eliminarea lor din setul de noduri nesetate al următoarei iterații și adăugarea lor la mulțimea de noduri de analizat a următoarei iterații. Fiecare fir de execuție GPU verifică distanța pentru nodul corespunzător și o actualizează dacă distanța este mai mică, marcând nodul ca fiind setat (Ortega-Arranz et al. 2013).

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 *Algoritmul Bellman – Ford paralelizat pe GPU*

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

<<<inițializare vector distanțe>>>(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 de distanțe sunt la fel sau se găsește un circuit de cost negativ.

Algoritmul 3.4 *Kernel 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

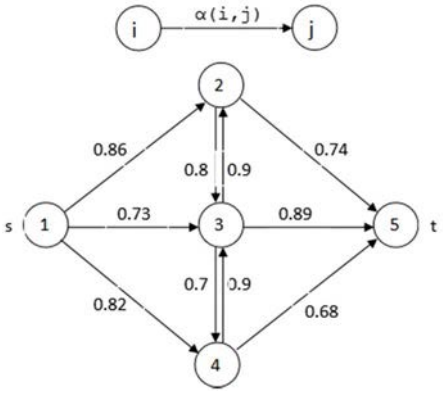


Figura 3.3 *Rețea generalizată, G = (V,E, β)*

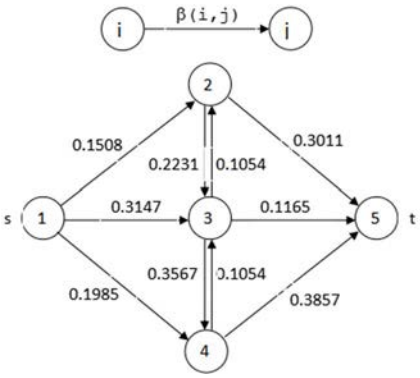


Figura 3.4 *Rețea modificată, G = (V,E, β)*

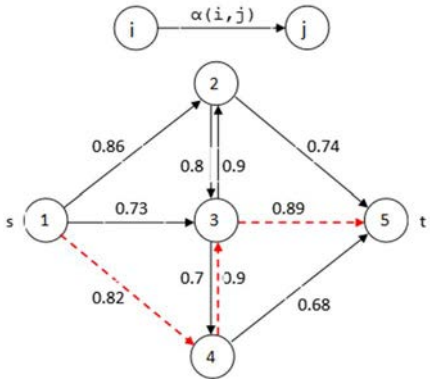


Figura 3.5 *Drumul cu pierderea minimă*

Pentru a ilustra modul în care **Algoritmul 3.1** tratează MLPP vom lua în considerare un exemplu. Astfel, în **Figura 3.3**, este dată o rețea generalizată. Un MLP trebuie calculat de la $s = 1$ la $t = 5$. Prin aplicarea logaritmului natural la valorile α (factorii de pierdere ai arcelor), se obține rețeaua din **Figura 3.4**. Folosind

valorile obținute, drumul cel mai scurt de la s la t este $P = (1,4,3,5)$ având lungimea de $0,1985 + 0,1054 + 0,1165 = 0,4204$. P este MLP în rețeaua inițială (**Figura 3.5**) având factorul de pierdere de $0,82 \times 0,9 \times 0,89 = 0,65682$, ceea ce înseamnă că dacă o unitate de flux este trimisă de la nodul s pe calea P, ajung doar 0,65682 unități de flux la nodul t. Deci, pierderea totală pe P este $1 - 0,65682 = 0,34318$. Este ușor de observat că orice alt drum de la s la t are o rata de livrare mai mare și, în consecință, pierderea pe un oricare astfel de drum este mai mare decât pierderea pe P.

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. În **Tabel 3.1** sunt prezentați timpii de execuție pe CPU și pe GPU și creșterea vitezei de execuție (speed-up) corespunzătoare **Algoritmul 3.1** bazat pe algoritmul lui Dijkstra. 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 2.7**. După cum se poate observa, 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 execuție crește până la 390 pentru rețelele cu 40000 de noduri și până la 326M de arce, așa cum se poate observa în **Figura 3.6**.

Tabel 3.1 *Timpii de execuție și speed-up pentru* ***Algoritmul 3.1***

Numărul de noduri	Numărul de arce	Timpul de execuție pe CPU (s)	Timpul de execuție pe GPU (s)	Speed-up
2000	266863	0.038868	0.316784	0.122696
	545296	0.058289	0.243916	0.238972
	792044	0.068866	0.209233	0.329135
	1024298	0.074267	0.186745	0.397692
5000	1730230	0.370748	0.266437	1.391503
	3260184	0.48296	0.185177	2.608099

	4667223	0.548423	0.204094	2.687109
	6193292	0.617939	0.22487	2.747983
10000	6780947	1.916246	0.420077	4.561654
	13011798	2.399974	0.406194	5.908443
	18582466	2.703393	0.363333	7.440538
	24094464	2.980178	0.283428	10.514762
15000	14574815	4.788783	0.430876	11.114063
	28771632	6.023874	0.248085	24.281492
	41488314	6.80067	0.269658	25.219612
	54199030	7.474292	0.252921	29.551884
20000	24619509	9.202524	0.371545	24.768262
	47347736	11.328984	0.216207	52.398784
	68784930	12.795397	0.218007	58.692597
	89281904	13.808837	0.196592	70.241093
25000	36956626	15.115443	0.331355	45.617066
	71570683	18.50168	0.239202	77.347514
	105252771	20.662288	0.203471	101.549056
	135731348	22.45409	0.186871	120.158238
30000	58783520	23.39317	0.239929	97.500386
	114142749	28.976564	0.201902	143.517964
	166982018	32.632028	0.221289	147.463399
	212557196	22.45409	0.186871	212.41397
35000	81732865	30.203684	0.258476	116.852953
	155029960	36.423584	0.19904	182.996302
35000	226714922	42.959844	0.196212	218.946058
	290748972	57.446978	0.287694	199.680834
40000	93255110	31.732728	0.229257	138.415525
	177326518	38.486862	0.198628	193.763528
	255241877	49.045823	0.211874	231.485803
	326548837	79.707052	0.221837	359.304589
50000	115598831	33.916584	0.245294	138.269114
	218427430	40.67372	0.202486	200.871764

Rețelele analizate au diferite dimensiuni, crescând de la 2000 de noduri până la 50000 de noduri. În mod evident, cu cât rețeaua este mai mare (adică are

mai multe noduri și arce), cu atât crește și timpul de execuție al algoritmului. În general, rezultatele arată că timpul de execuție pe GPU este mult mai mic decât timpul de execuție pe CPU pentru rețelele de toate dimensiunile. Aceasta demonstrează beneficiile paralelizării cu CUDA pentru algoritmul Dijkstra. În timp ce timpul de execuție pe CPU crește semnificativ odată cu creșterea dimensiunii rețelei, timpul de execuție pe GPU crește într-un ritm mult mai lent. Speed-up-ul, calculat ca raport între timpul de execuție pe CPU și timpul de execuție pe GPU, arată cu cât este mai rapidă execuția în cazul paralelizării pe GPU în comparație cu implementarea secvențială, pe CPU (**Figura 3.6**).

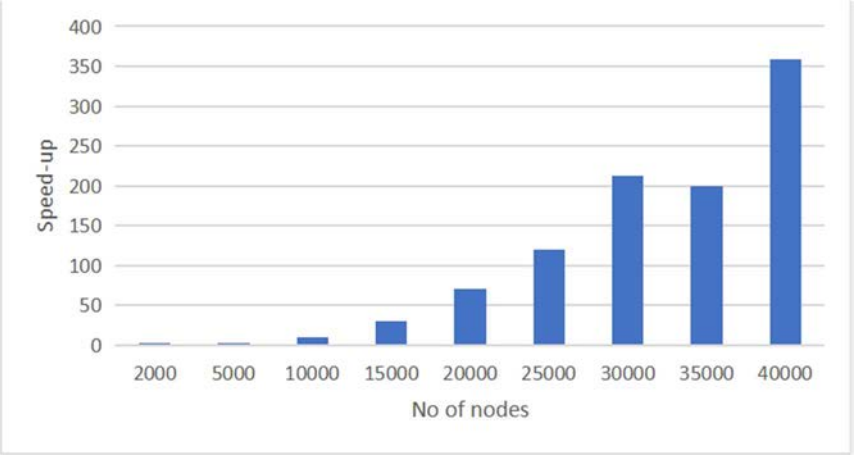


Figura 3.6 *Creșterea vitezei de execuție (speed-up) pe GPU pentru* **Algoritmul 3.1** *pe rețele dense cu număr variat de noduri* (Deaconu, **Spridon** și Ciupala 2023)

Valorile speed-up-ului sunt semnificative, crescând odată cu creșterea dimensiunii rețelelor. Acest lucru sugerează că beneficiile paralelizării cu CUDA devin mai pronunțate pentru rețelele mai mari. Rezultatele indică faptul că paralelizarea cu CUDA aduce îmbunătățiri semnificative de performanță pentru **Algoritmul 3.1**, în special pentru rețelele mai mari.

Tabel 3.2 *Timpii de execuție și speed-up pentru* ***Algoritmul 3.2***

Numărul de noduri	Numărul de arce	Timpul de execuție pe CPU (s)	Timpul de execuție pe GPU (s)	Speed-up
2000	266863	0.008697	0.006947	1.251907
	545296	0.014915	0.012184	1.224146

	792044	0.023982	0.017236	1.39139
	1024298	0.031349	0.021302	1.471646
5000	1730230	0.056682	0.018686	3.033394
	3260184	0.102837	0.032626	3.151995
	4667223	0.135491	0.038629	3.507494
	6193292	0.159503	0.055479	2.875016
10000	6780947	0.260996	0.051385	5.079225
	13011798	0.45296	0.096248	4.706176
	18582466	0.603817	0.118198	5.108521
	24094464	0.832522	0.142914	5.825336
15000	14574815	0.520198	0.105376	4.936589
	28771632	1.077742	0.220179	4.894845
	41488314	1.38311	0.306078	4.518815
	54199030	1.76405	0.405787	4.347231
20000	24619509	0.868734	0.199072	4.363919
	47347736	1.671984	0.415297	4.025996
	68784930	2.275151	0.606868	3.749005
	89281904	2.634883	0.741772	3.552147
25000	36956626	1.289436	0.34979	3.686315
	71570683	2.295006	0.691613	3.318338
	105252771	3.33781	0.969819	3.441683
	135731348	3.966533	1.301746	3.047087
30000	58783520	2.185981	0.65262	3.349546
	114142749	4.426504	1.381369	3.204433
	166982018	5.348372	1.952395	2.73939
	212557196	7.050751	2.54587	2.769486
35000	81732865	2.736643	0.975467	2.80547
	155029960	5.009507	1.899943	2.636662
	226714922	6.539598	2.522306	2.592706
	290748972	8.390887	3.381186	2.48164
40000	93255110	3.143473	1.109501	2.833231
	177326518	5.472819	2.072796	2.640308
	255241877	7.791466	2.861063	2.723277
	326548837	10.25897	4.578939	2.240469
50000	115598831	4.535481	1.671546	2.713345

	218427430	7.247106	2.805613	2.583074

În **Tabel 3.2,** timpii de execuție și accelerațiile sunt prezentate pentru **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 24000 de arce (**Figura 3.7**).

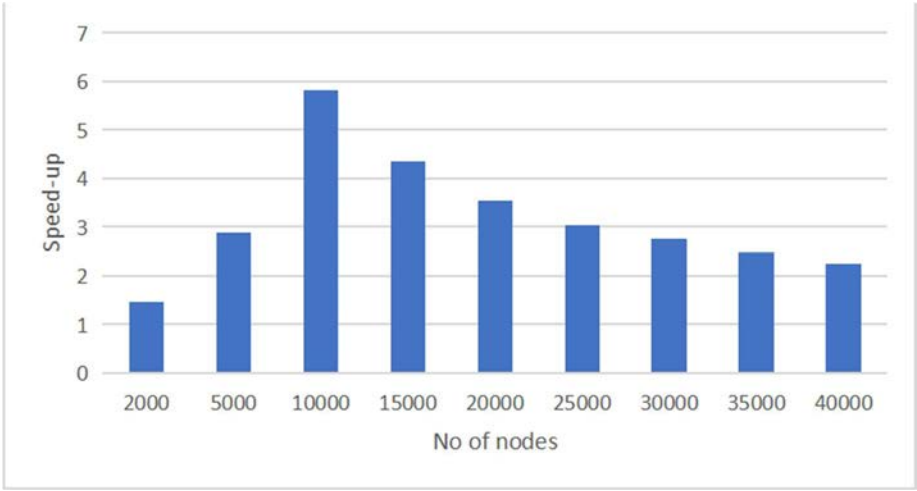


Figura 3.7 *Creșterea vitezei de execuție (speed-up) pe GPU pentru* ***Algoritmul 3.2*** *pe rețele dense cu număr variat de noduri* (Deaconu, **Spridon** și Ciupala 2023)

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 3.8**).

De asemenea, odată cu creșterea densității rețelei crește și timpul de execuție pentru implementarea secvențială, în timp ce, pentru implementarea paralelă se observă o creșterea mai lentă a timpului de execuție odată cu creșterea densității rețelei (**Figura 3.9**).

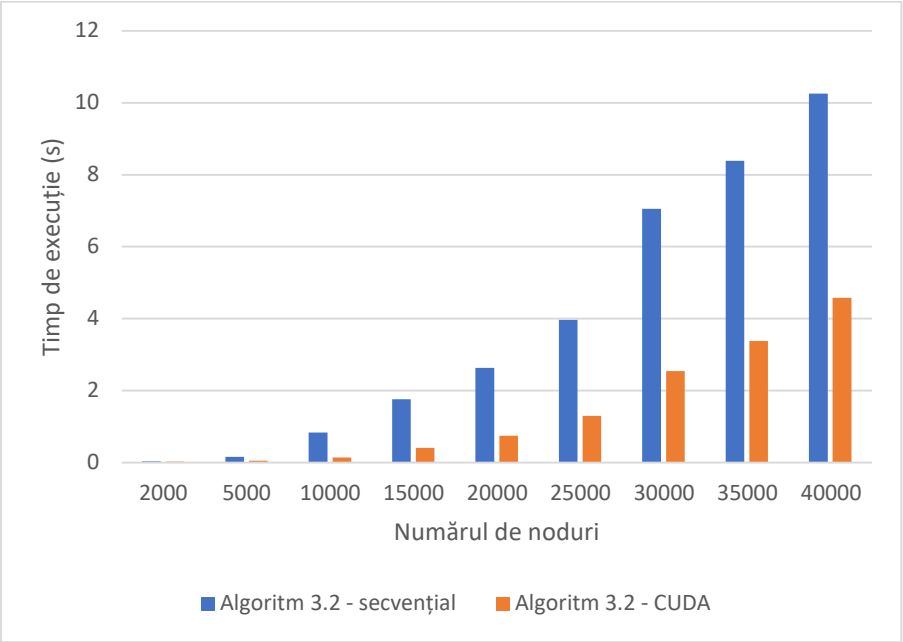


Figura 3.8 *Timpul de execuție pentru Algoritmul 3.2 pentru rețele dense*

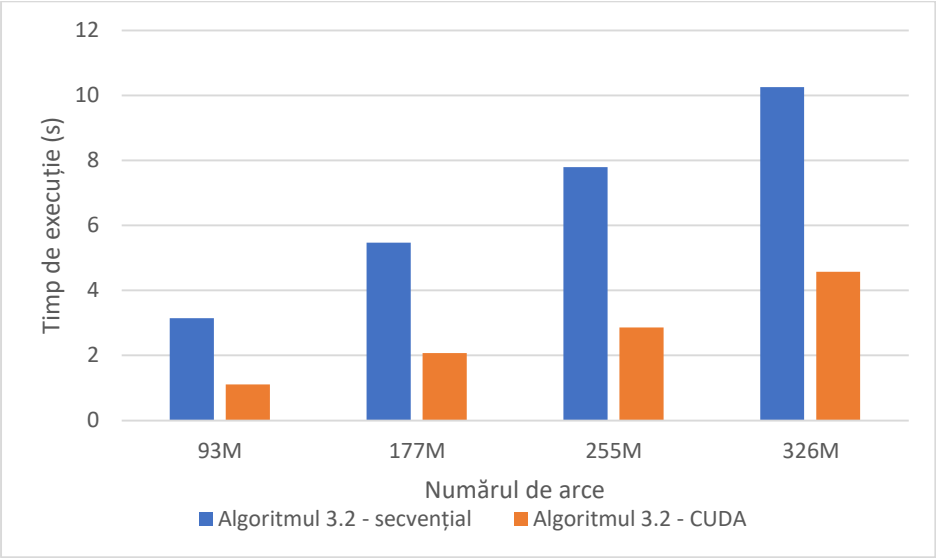


Figura 3.9 *Evoluția timpului de execuție pentru **Algoritmul 3.2** în funcție de densitatea rețelei*

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 acestora. 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 3.8**). 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 se prezintă 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. Se prezintă 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.

4.1. Problema tradițională a fluxului maxim

Această problemă a fost studiată pentru prima dată de Dantzig în 1951 (Dantzig, Applications of the simplex method to a transportation problem. 1951) și Ford împreună cu Fulkerson în anul 1956 (Ford și Fulkerson 1956). Problema constă găsirea unui mod de a transporta o cantitate cât mai mare de flux de la un nod, numit sursă, s, la un alt nod, numit stoc, t, ținând cont de capacitățile maxime admise pe fiecare arc. De exemplu, o companie de furnizare de electricitate poate dori să maximizeze cantitatea de energie electrică trimisă între o pereche de orașe prin rețeaua sa, fiecare traseu electric din rețea având o capacitate limitată.

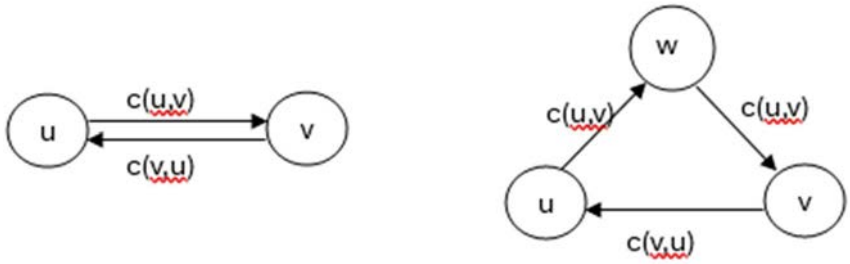
Cu alte cuvinte, în problema fluxului maxim, obiectivul este de a trimite cât mai mult flux posibil între două noduri, respectând limitele de capacitate ale arcelor. O instanță a problemei fluxului maxim este o rețea $G = (V, E, s, t, c)$ antisimetrică, unde $s \in V$ este nodul sursă, $t \in V$ este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

o funcție de capacitate. Pentru rezolvarea acestei probleme sunt necesare câteva noțiuni teoretice pe care le voi expune în cele ce urmează.

O rețea este *antisimetrică* dacă satisface condiția:

$$\forall (u, v) \in E: (v, u) \notin E \tag{4.1}$$

Pentru o rețea care nu este antisimetrică se poate construi o rețea antisimetrică echivalentă prin introducerea unor noduri w în rețea. De exemplu, rețeaua din **Figura 4.1 a)** nu este antisimetrică, întrucât există arc atât între nodurile u și v cât și între nodurile v și u. Adăugând nodul w în rețea și construind arcele (u, w) și (w, v), rețeaua din **Figura 4.1 b)** a devenit antisimetrică.



a) b)

Figura 4.1 a) *Rețea care nu este antisimetrică,* **b)** *Rețea antisimetrică echivalentă*

Astfel, algoritmul de determinare a fluxului maxim se poate aplica pentru orice rețea antisimetrică, sau, în cazul în care nu este antisimetrică pe echivalenta antisimetrică a acesteia.

Într-o rețea, un *pseudo-flux* este o funcție care satisface constrângerile de capacitate (4.2) și pe cele de antisimetrie (4.3):

$$\forall (u, v) \in E: f(u, v) \leq c(u, v) \tag{4.2}$$
$$\forall (u, v) \in E: f(u, v) = -f(v, u) \tag{4.3}$$

Un flux este un pseudo-flux care satisface și constrângerea de conservare a fluxului:

$$\forall v \in V - \{s, t\}: \sum_{u \in V:(v,u) \in E} f(u, v) = 0 \tag{4.4}$$

Altfel spus, pentru toate nodurile cu excepția sursei și a stocului, fluxul net este zero. Nu trebuie să facem distincție între fluxul care intră și iese din nodul v din cauza constrângerilor de antisimetrie. Valoarea unui flux f este fluxul net în nodul stoc:

$$|f| = \sum_{v \in V:(v,t) \in E} f(u,v) \tag{4.5}$$

Un *drum de mărire a fluxului* este un drum $s - t$ în rețeaua reziduală, G_f. Este evident că, dacă există un drum de mărire în G_f, atunci putem mări f trimițând fluxul de-a lungul acestui drum. Ford și Fulkerson (Ford și Fulkerson 1956) au arătat că reciproca este, de asemenea, adevărată (**Teorema 4.1**).

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 $c_f: E \rightarrow \mathbb{R}$, $c_f(u,v) = c(u,v) - f(u,v)$, 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 (**Algoritmul 4.1)** (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. **Algoritmul 4.1** este o variantă simplificată a algoritmului propus de Ford și Fulkerson, pentru

găsirea unui drum de mărire fiind posibile diferite abordări (prin etichetare, prin parcurgere în lățime, etc.).

Algoritmul 4.1 *Ford – Fulkerson pentru determinarea fluxului maxim*

Parametri de intrare:

- O rețea $G = (V, E, c, s, t)$

Parametri de ieșire:

- f – fluxul maxim prin G

Se consideră un flux admisibil, $f = 0$
Se inițializează rețeaua reziduală $G_f = G$
Cât timp există drum de mărire de s la t în G_f
    Se găsește un drum de mărire, P, în G_f
    Se determină $r = \min\{c(u,v) | (u,v) \in P\}$
    Se crește valoarea fluxului, $f = f + r$
    Se actualizează rețeaua reziduală, G_f
Sfârșit cât timp

Algoritmul începe cu un flux admisibil $f = 0$, iar rețeaua reziduală, G_f, este aceeași cu rețeaua inițială. Iterativ, se găsește un drum de la s la t care mai are capacitate disponibilă, $P \in G_f$. Se determină capacitatea minimă, r, dintre cele ale arcelor care aparțin drumului nou găsit și se mărește fluxul de-a lungul acelui drum cu cantitatea maximă posibilă, r. Pasul următor presupune actualizarea rețelei reziduale de-a lungul respectivului drum. Actualizarea se face conform formulelor (4.6), pentru $\forall (u,v) \in P$. Acest proces continuă până când nu mai pot fi găsite drumuri de mărire în rețeaua reziduală.

$$\begin{cases} c_f(u,v) = c(u,v) - r \\ c_f(v,u) = c(v,u) + r \end{cases} \tag{4.6}$$

4.2. Problema fluxului maxim generalizat

Problema fluxului maxim generalizat este o extindere a problemei tradiționale a fluxului maxim. Aceasta a fost studiată pentru prima dată în (Dantzig, Linear programming and extensions 1962) și (Jewell, Optimal flow through networks with gains 1962). În rețelele tradiționale, există o presupunere implicită că fluxul este conservat pe fiecare arc. Această presupunere poate fi încălcată dacă, de exemplu, gazul natural se scurge în timp ce este pompat printr-o conductă.

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 $\alpha : E \rightarrow \mathbb{R}_{>0}$. Pentru fiecare unitate de flux care intră în arcul (u, v) 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).

Astfel, determinarea fluxului maxim într-o rețea cu pierderi pe arce este un concept avansat care poate fi aplicat în diverse domenii unde fluxul de resurse sau informații este afectat de pierderi datorate frecării, rezistenței, degradării sau alte forme de reducere a capacității de transport. Câteva aplicații concrete în viața reală sunt:

- În rețelele de distribuție a apei, pierderile de apă din conducte (datorate scurgerilor, evaporării sau defectelor) sunt comune. Determinarea fluxului maxim cu pierderi ajută la optimizarea transportului apei de la surse la consumatori, minimizând pierderile și asigurând o distribuție eficientă.
- În rețelele de transport și distribuție a energiei electrice, există pierderi de energie datorate rezistenței conductorilor și altor factori. Algoritmii de flux maxim cu pierderi pot optimiza fluxul de energie, asigurând o distribuție eficientă și minimizând pierderile de energie.

- În rețelele de distribuție a gazelor naturale, pierderile de presiune și scurgerile sunt frecvente. Determinarea fluxului maxim cu pierderi ajută la planificarea și operarea eficientă a rețelei pentru a asigura livrarea optimă a gazelor la destinații.
- În rețelele de transport, vehiculele suferă pierderi de energie din cauza frecării și rezistenței aerodinamice. Optimizarea rutelor și încărcăturii poate fi realizată folosind modele de flux maxim cu pierderi pentru a minimiza consumul de combustibil și timpul de transport.
- În rețelele de telecomunicații și internet, semnalele pot pierde putere datorită atenuării pe distanțe mari sau interferențelor. Algoritmii de flux maxim cu pierderi pot optimiza rutele de transmisie pentru a asigura o calitate optimă a serviciului și a minimiza pierderile de date.
- În sistemele de canalizare, pierderile pot apărea din cauza scurgerilor sau colmatării conductelor. Determinarea fluxului maxim cu pierderi ajută la proiectarea și întreținerea rețelei pentru a asigura o evacuare eficientă a apelor uzate.
- Sistemele de irigații suferă pierderi de apă din cauza evaporării și infiltrației în sol. Optimizarea fluxului de apă cu pierderi ajută la gestionarea eficientă a resurselor de apă pentru agricultură.
- În rețelele de conducte pentru petrol și produse petroliere, pierderile pot apărea din cauza scurgerilor și evaporării. Determinarea fluxului maxim cu pierderi ajută la optimizarea transportului pentru a minimiza pierderile și a asigura o distribuție eficientă.

În toate aceste domenii, algoritmii de determinare a fluxului maxim cu pierderi pe arce sunt esențiali pentru a optimiza utilizarea resurselor, a minimiza pierderile și a îmbunătăți eficiența operațională a rețelelor respective.

În problema generalizată a fluxului maxim, obiectivul este să se trimită cât mai mult flux posibil între două noduri, ținând cont de constrângerile de capacitate ale arcelor și de faptul că pe fiecare arc pot exista pierderi sau câștig de flux.

Un exemplu concret este rețeaua din **Figura 4.2** în care nodul sursa este S, iar nodul stoc este, T. Astfel, dacă pe arcul (S, A) care are rata de livrare $\alpha(S, A) = 0.9$ trimitem prin nodul S 10 unități de flux, în nodul A vor ajunge 9 unități de flux. Acest lucru poate fi privit și ca trimițând -9 unități de flux prin nodul A către nodul S prin arcul (A, S) care are factorul de câștig / pierdere $\alpha(A, S) = 1 / 0.9$, astfel

ajungând în nodul S, -10 unități de flux. Ca și în problema anterioară, fluxurile negative sunt introduse ca notație pentru a face distincție între sensurile în care curge fluxul.

Putem considera, ca exemplu practic, o rețea de distribuție a energiei electrice în care energia electrică este generată în centrale și trebuie distribuită către consumatori prin intermediul unei rețele de linii de transport. În această rețea, fiecare linie de transport are o anumită eficiență (un factor de conversie) datorită pierderilor de energie pe parcursul transmisiei. Nodurile reprezintă în acest caz centrale electrice, stații de transformare, și puncte de consum, iar arcele reprezintă liniile de transport, fiecare având o capacitate maximă (cantitatea maximă de energie care poate fi transmisă) și un factor de conversie (eficiența liniei).

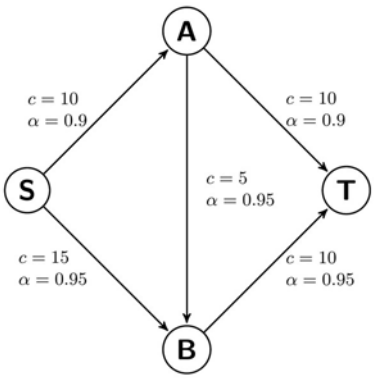


Figura 4.2 Exemplu de rețea cu pierderi

În acest context, un pseudo-flux în rețeaua generalizată este o funcție care satisface constrângerile de capacitate (4.2) și pe cele de antisimetrie într-o astfel de rețea:

$$\forall (u, v) \in E: f(u, v) = -\alpha(v, u) \cdot f(v, u) \tag{4.7}$$

Un flux într-o rețea cu pierderi este un pseudo-flux care nu are deficite reziduale în rețeaua respectivă, dar este permis să aibă surplusuri.

Pentru determinarea rețelei reziduale, trebuie să se țină cont de rata de livrare (factorul de câștig / pierdere) de pe fiecare arc. Astfel, dacă considerăm un flux f pe arcul (u, v), atunci funcția de capacitate rezidual este:

$$\begin{cases} c_f(u,v) = c(u,v) - f/\alpha(u,v) \\ c_f(v,u) = c(v,u) + f \end{cases} \quad (4.8)$$

Totodată, dacă rata de livrare pe arcul (u, v) este $\alpha(u, v)$, se poate considera că rata de livrare pe arcul (v, u) este $\alpha(v, u) = 1 / \alpha(u, v)$.

Pentru a rezolva problema generalizată a fluxului maxim pentru determinarea fluxului cu pierdere minimă, se propune o adaptare a algoritmului Ford – Fulkerson (**Algoritmul 4.2**), astfel încât, la fiecare iterație, pentru găsirea unui nou drum în rețeaua rezidual 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 rezidual 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 rata de livrare $\alpha(u, v)$, atunci rata de livrare în rețeaua rezidual 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 4.2 *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 rezidual $G_f = G$
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 4.3**

Sfârșit cât timp

Pentru actualizarea rețelei reziduale, G_f, de-a lungul drumului, P, găsit se folosește **Algoritmul 4.3.** Acest algoritm implică determinarea fluxului admisibil maxim pe drumul de mărire și apoi actualizarea capacităților reziduale și a ratelor de livrare 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 rata de livrare $\alpha_f(u,v)$.
     - 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 4.3 *Actualizarea rețelei reziduale după găsirea unui nou drum de mărire în* $\boldsymbol{G_f}$

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

Pentru a înțelege mai bine algoritmul propus, este prezentat un exemplu în **Figura 4.3.**

Astfel, în **Figura 4.3** rețeaua reziduală inițială este $G_f = G$, nodul sursă fiind S, iar cel stoc, T. În iterația 1 drumul cu pierderi minime, găsit în G_f, este drumul $P = S{-}{>}\ B{-}{>}\ T$ (rata de livrare este: $\alpha_f(S,B) \cdot \alpha_f(B,T) = 0.2 \cdot 0.5 = 0.1$). Pe acest drum se pornește cu fluxul $f = c_f(S,B) = 40$. Pe arcul (S,B) fluxul $f = \min\{f \cdot \alpha_f(S,B) = 8,\ c_f(S,B) \cdot \alpha_f(S,B) = 8\}$. Astfel, în nodul B ajung 8 unități de flux. Pe arcul (B,T) fluxul $f = \min\{f \cdot \alpha_f(B,T) = 4,\ c_f(B,T) \cdot \alpha_f(B,T) = 2\}$. În nodul T ajung 2 unități de flux. Conform **Algoritmul 4.3** rețeaua reziduală actualizată se determină pornind cu fluxul f determinat pe drumul cu pierdere minimă găsit, în sens invers, de la nodul stoc, către nodul sursă. Astfel, pornind de la T, arcul (T,B) va avea capacitatea $c_f(T,B) = c_f(T,B) + f = 2$, iar $c_f(B,T) = 0$. Rata de livrare $\alpha_f(T,B) = \frac{1}{0,5} = 2$, deci în

nodul B intră $f \cdot \alpha_f(T,B) = 4$ unități de flux. Pentru arcul (S,B), $c_f(S,B) = c_f(S,B) - \frac{4}{0.2} = 20$, iar $c_f(B,S) = 4$ și $\alpha_f(B,S) = \frac{1}{0.2} = 5$.

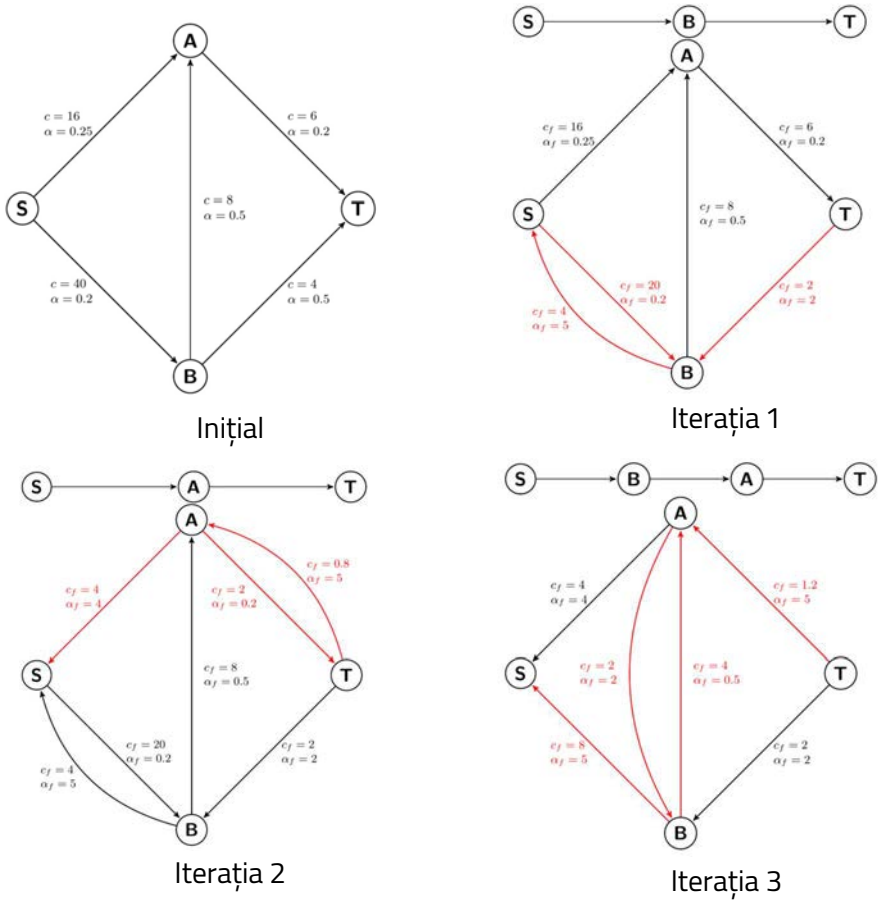


Figura 4.3 *Exemplu de determinare a fluxului maxim într-o rețea cu pierderi*

În iterația 2 drumul cu pierderi minime, găsit în G_f este $P = S{-}{>}\ A{-}{>}\ T$ (rata de livrare este: $c_f(S,A) \cdot \alpha_f(A,T) = 0.1 \cdot 0.2 = 0.02$). Pe acest drum se pornește cu fluxul $f = c_f(S,A) = 16$. Pe arcul (S,A) fluxul $f = \min\{f \cdot \alpha_f(A,T) = 4,\ c_f(S,A) \cdot \alpha_f(S,A) = 4\}$. Astfel, în nodul A ajung $16 \cdot 0.25 = 4$ unități de flux. Pe arcul (A,T) fluxul $f = \min\{f \cdot \alpha_f(A,T) = 0.8,\ c_f(A,T) \cdot \alpha_f(A,T) = 1.2\}$. În nodul T ajung 0.8 unități de flux. Se actualizează rețeaua reziduală pornind cu fluxul $f = 0.8$. Astfel, pornind de la T, arcul (T,A) va avea capacitatea $c_f(T,A) = c_f(T,A) + f = 0.8$, iar $c_f(A,T) = 2$. Rata de livrare

$\alpha_f(T,A) = \frac{1}{0.2} = 5$, deci în nodul A intră $f \cdot \alpha_f(T,A) = 4$ unități de flux. Pentru arcul (S,A), $c_f(S,A) = c_f(S,A) - \frac{4}{0.25} = 0$, iar $c_f(A,S) = 4$ și $\alpha_f(A,S) = \frac{1}{0.25} = 4$.

În iterația 3 drumul cu pierderi minime, găsit în G_f este $P = S{-}{>}\ B{-}{>}\ A{-}{>}\ T$ (factorul de pierdere este: $\alpha_f(S,B) \cdot \alpha_f(B,A) \cdot \alpha_f(A,T) = 0.2 \cdot 0.5 \cdot 0.2 = 0.02$). Pe acest drum se pornește cu fluxul $f = c_f(S,B) = 20$. Pe arcul (S,A) fluxul $f = \min\{f \cdot \alpha_f(S,B) = 4,\ c_f(S,B) \cdot \alpha_f(S,B) = 4\}$. Astfel, în nodul B ajung 4 unități de flux. Pe arcul (B,A) fluxul $f = \min\{f \cdot \alpha_f(B,A) = 2,\ c_f(B,A) \cdot \alpha_f(B,A) = 4\}$. În nodul A ajung 2 unități de flux. Pe arcul (A,T) fluxul $f = \min\{f \cdot \alpha_f(A,T) = 0.4,\ c_f(A,T) \cdot \alpha_f(A,T) = 0.4\}$. În nodul T ajung 0.4 unități de flux. Se actualizează rețeaua reziduală pornind cu fluxul $f = 0.4$. Astfel, pornind de la T, arcul (T,A) va avea capacitatea $c_f(T,A) = c_f(T,A) + f = 1.2$, iar $c_f(A,T) = 0$. În nodul A intră $f \cdot \alpha_f(T,A) = 2$ unități de flux. Pentru arcul (B,A), $c_f(B,A) = c_f(B,A) - \frac{2}{0.5} = 4$, iar $c_f(A,B) = 2$ și $\alpha_f(A,B) = \frac{1}{0.5} = 2$. Pentru arcul (S,B), $c_f(S,B) = c_f(S,B) - \frac{4}{0.2} = 0$, iar $c_f(B,S) = 8$.

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 4.2** complexitatea acestuia este dată de complexitatea **Algoritmul 3.2** înmulțită cu numărul de iterații în care se determină câte un drum de mărire. În **Tabel 4.1** sunt prezentate rezultatele experimentale obținute folosind același sistem descris în **Capitolul 3**.

Tabel 4.1 *Timpii de execuție și speed-up pentru* **Algoritmul 4.2**

Numărul de noduri	Numărul de arce	Timpul de execuție pe CPU (s)	Timpul de execuție pe GPU (s)	Speed-up

2000	266863	0.08897	0.085364	1.042243
	545296	0.2993	0.269048	1.112441
	792044	0.72046	0.585024	1.231505
	1024298	1.25296	0.95729	1.308861
5000	1730230	2.26628	0.8387	2.702134
	3260184	6.1682	2.075064	2.972535
	4667223	9.50437	2.87175	3.309609
	6193292	12.7624	4.712715	2.708078
10000	6780947	26.096	5.43481	4.80164
	13011798	58.9848	12.6801	4.651762
	18582466	90.47255	18.02492	5.019304
	24094464	140.5387	24.57208	5.719448
15000	14574815	78.0297	15.6064	4.999853
	28771632	153.2161	37.87079	4.045761
	41488314	248.9598	55.9062	4.45317
	54199030	345.1695	78.51689	4.894848526
20000	24619509	155.05946	38.430896	4.03476047
	47347736	334.5968	84.30291	3.968982803
	68784930	500.3322	135.3364	3.696952187
	89281904	632.392	179.08824	3.53117547
25000	36956626	386.6308	103.686	3.728864
	71570683	767.352	223.0736	3.439905
	105252771	1274.99	356.5916	3.57549
	135731348	1516.948	509.3967	2.977931

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.

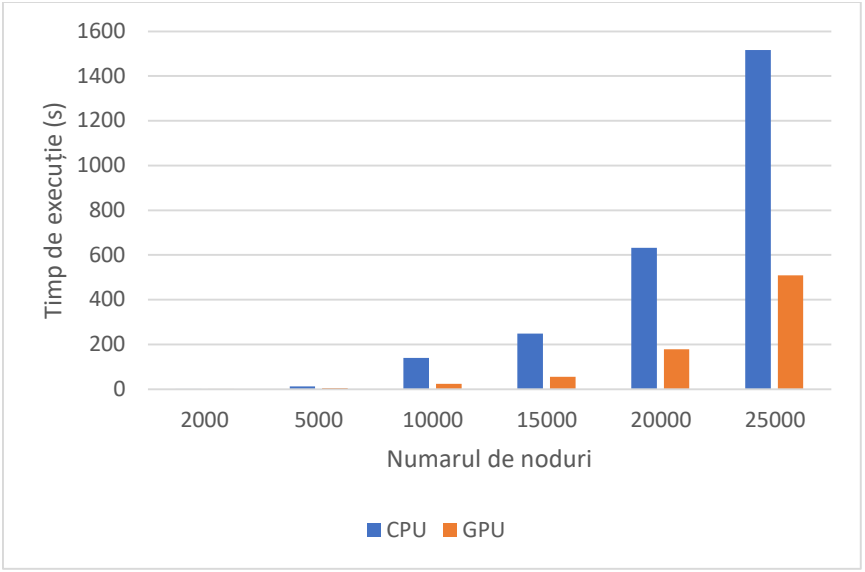


Figura 4.4 Timpi de execuție pentru **Algoritmul 4.2** în rețele dense

După cum se poate observa în **Figura 4.4** 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

În capitolul curent se prezintă metode GPU pentru obținerea de hărți de poluare și geomagnetice folosind metode de interpolare, pornind de la măsurători în diferite puncte dintr-o anumită zonă geografică. Sunt realizate totodată, analize de precizie a hărților obținute și de eficiență a metodelor GPU utilizate.

5.1. Metode de interpolare bi-dimensională

De-a lungul timpului au fost dezvoltate mai multe metode de interpolare și aproximare pentru a estima valorile anumitor parametri în locații cu valori necunoscute ale acestora (Franke 1985, Burrough 1986). Aceste metode au fost concepute pentru a fi posibile transformări între diferite reprezentări discrete și continue ale câmpurilor spațiale și spațio - temporale, de obicei pentru a transforma date neregulate de punct sau linie în reprezentare raster sau pentru a modifica rezoluția unei hărți date.

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-ul sau formele funcționale ad-hoc (polinoame, multi-quadric). Alegerea condiției suplimentare depinde de caracterul fenomenului modelat și de tipul aplicației.

Metodele de interpolare fiabile ar trebui să satisfacă mai multe condiții importante: acuratețe și putere predictivă, robustețe și flexibilitate în descrierea diferitelor tipuri de fenomene, netezire pentru date zgomotoase, formulare d-dimensională (pentru cazuri generale de spații d - dimensionale), estimarea directă a derivatelor (gradienți, curburi) , aplicabilitate la seturi mari de date, eficiență de calcul și ușurință în utilizare. Este dificil să găsești o metodă care să îndeplinească toate cerințele menționate mai sus pentru o gamă largă de date georeferențiate. Prin urmare, alegerea unei metode adecvate cu parametri potriviți pentru o anumită aplicație este crucială. Metode diferite pot duce la reprezentări spațiale destul de diferite și este necesară cunoașterea aprofundată a fenomenului pentru a evalua care dintre ele este cea mai apropiată de realitate. Utilizarea unei metode nepotrivite sau a unor parametri nepotriviți poate avea ca rezultat un model distorsionat de distribuție spațială, ceea ce duce la decizii potențial greșite bazate pe informații spațiale înșelătoare.

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

5.1.1. Ponderarea folosind inversul distanței

Metoda IDW (Inverse Distance Weighting) este un algoritm de interpolare adoptat pe scară largă, utilizat pentru a estima valorile în locații în care nu exista măsurători, pe baza valorilor eșantion, măsurate în locații din apropierea acestora. Algoritmul presupune că valoarea estimată este o funcție de distanța dintre punctul în care se face estimarea și locațiile punctelor eșantion, 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.

Ideea de bază din spatele IDW este de a calcula o medie ponderată a valorilor măsurate în diferite puncte, ponderile fiind proporționale cu inversul distanței dintre punctul în care se realizează estimarea și fiecare punct eșantion. Ponderile sunt de obicei normalizate astfel încât suma lor să fie unu. Valoarea estimată într-un anumit punct este media ponderată a valorilor măsurate.

Algoritmul IDW poate fi exprimat matematic după cum urmează:

Fie $V(u)$ valoarea estimată într-un anumit punct, u, $V(i)$ este valoarea măsurată în al i-lea punct eșantion, care are coordonatele $r_i = (x_i, y_i)$, iar este $l(i)$ distanța dintre punctul de estimat și al i-lea punct eșantion. În acest caz, valoarea estimată într-un anumit punct poate fi calculată folosind următoarea formulă:

$$V(u) = \frac{\sum_{i=1}^{k} w(i) * V(i)}{\sum_{i=1}^{k} w(i)} \quad (5.2)$$

unde $w(i) = \frac{1}{l(i)^p}$, p este un exponent pozitiv care controlează rata la care ponderile scad odată cu distanța și k este numărul total de puncte în care sunt valori măsurate.

Există diferite modalități de a alege valoarea exponentului p, dar cele mai frecvente valori sunt $p = 2$ (ponderarea în funcție de inversul distanței la pătrat) și $p = 3$ (ponderarea în funcție de inversul distanței cubice).

IDW este un algoritm simplu și eficient din punct de vedere computațional, dar are unele limitări. Una dintre principalele limitări este că se presupune că valorile estimate sunt doar o funcție de distanță și nu iau în considerare alți factori care pot influența valorile, cum ar fi topografia, tipul de sol sau utilizarea terenului. O altă limitare este că poate produce estimări nerealiste în locații care se află la distanță mare de toate punctele eșantion, mai ales atunci când locațiile acestora sunt distribuite inegal.

În ciuda limitărilor sale, IDW este utilizat pe scară largă în diverse domenii, inclusiv geologie, știința mediului și geografie. Este adesea folosit ca metodă de bază pentru compararea cu alți algoritmi de interpolare și este, de asemenea, utilizat pentru aplicații simple în care nu este necesară o precizie ridicată.

5.1.2. Kriging

Ideea de bază din spatele metodei kriging de interpolare este să se estimeze valoarea necunoscută $z(u)$ 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.

Estimarea kriging pentru valoarea necunoscută $V(u)$ este dat de:

$$V(u) = \sum_{i=1}^{n} w_i V(x_i) \quad (5.3)$$

unde w_i sunt ponderile de kriging și $V(x_i)$ sunt valorile observate în locațiile punctelor eșantion x_i . Ponderile de kriging se obțin prin rezolvarea următorului sistem de ecuații:

$$Cw = \lambda \quad (5.4)$$

unde C este matricea de autocovarianță spațială dintre valorile măsurate, x_i, w este vectorul ponderilor de kriging și λ este un multiplicator Lagrange care impune condiție de imparțialitate asupra estimatorului.

Algoritmul kriging poate fi extins pentru a încorpora informații suplimentare, cum ar fi tendința și eroarea de măsurare, prin modificarea sistemului kriging al ecuațiilor. Există mai multe variante de kriging, inclusiv kriging obișnuit, kriging simplu și kriging universal, fiecare cu ipoteze diferite despre structura de corelație spațială subiacentă.

Kriging-ul a fost aplicat pe scară largă în multe domenii, inclusiv geologie, hidrologie, ecologie și științe ale mediului și s-a dovedit a fi un instrument puternic pentru predicția spațială și cuantificarea incertitudinii (Cressie 1993) (Journel și Huijbregts 1978).

Câteva cercetări recente arată avantajele pe care CUDA le poate aduce atunci când este utilizat pentru paralelizarea metodelor de interpolare. Astfel, Liu, Hu și Tian (2021) au propus un algoritm paralel bazat pe CUDA pentru interpolarea IDW pe seturi de date mari, iar rezultatele au arătat că algoritmul bazat pe CUDA a depășit algoritmul tradițional bazat pe CPU în ceea ce privește timpul de calcul și precizia (Liu, Hu și Tian 2021). Un alt exemplu este prezentat de Gao, Cao și Yang în 2020 care au propus un algoritm paralel bazat pe CUDA pentru interpolarea Kriging și au ajuns la concluzia că algoritmul bazat pe CUDA a obținut accelerații semnificative în comparație cu algoritmul tradițional bazat pe CPU, în special pentru seturi mari de date (Gao, Cao și Yang 2020).

Ji, Wu și Wang (2020) au propus un cadru bazat pe CUDA pentru interpolarea IDW și Kriging și au arătat, de asemenea, îmbunătățirea performanței timpului și a preciziei (Ji, Wu și Wang 2020).

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 medie ponderată, $V(x, y)$, care se calculează utilizând ecuația (5.2), iar distanța dintre două puncte pe grid a fost calculată folosind formula pentru distanța pe Pământ între două puncte de anumite coordonate GPS (5.5).

$$\begin{aligned} a &= sin^2(\Delta\varphi/2) + cos\,\varphi 1 \cdot cos\,\varphi 2 \cdot sin^2(\Delta\lambda/2) \\ c &= 2 \cdot atan2(\sqrt{a}, \sqrt{1-a}) \\ d &= R \cdot c \end{aligned} \tag{5.5}$$

unde φ este latitudinea, λ este longitudinea, iar R - este raza Pământului (raza medie R = 6371 km).

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:

Pasul 1. Calculul punctelor de semi-variantă (**Algoritmul 5.2**)

Pasul 2. Calculul coeficienților de semi-variantă folosind metoda celor mai mici pătrate (Oliver and Webster 2014)

Pasul 3. Calculul ponderilor de interpolare (**Algoritmul 5.3**)

Pasul 4. Calculul valorilor interpolate (**Algoritmul 5.4**)

Algoritmul 5.2 *Calcularea semi-variației punctelor eșantion*

Parametri de intrare:

- Distanța maximă dintre punctele de influență: maxdist = 100km;
- Toleranța: toler = 2
- np - numărul de puncte eșantion
- P_i – coordonatele celui de-al i-lea punct eșantion ($i = 0, np - 1$)
- v_i – valoarea măsurată în cel de-al i-lea punct eșantion ($i = 0, np - 1$)

Parametri de ieșire:

- ns – numărul de puncte de semi-variantă $ns = \frac{maxdist}{toler}$
- s_k – valorile semi-variațelor $k = 0, np - 1$
- w_k- ponderi $k = n, ns - 1$

$ns = \frac{maxdist}{toler}$

Pentru $i = 0, ns - 1$ **execută**

$s_k = 0$

$w_k = 0$

sfârșit pentru

/ calcularea punctelor de semi-variantă */*

Pentru $i = 0, np - 2$ **execută**

Pentru $j = i + 1, np - 1$ **execută**

$d = dist(P_i, P_j)$

Dacă $d < maxdist + toler$ **atunci**

$k = \frac{d}{toler}$

$s_k = s_k + (v_i - v_j)^2$

$w_k = w_k + 1$

sfârșit dacă

sfârșit pentru

---

sfârșit pentru

Pentru $i = 0, np - 1$ **execută**

Dacă $w_k > 0$ **atunci**

$$s_k = \frac{s_k}{2w_k}$$

sfârșit dacă

sfârșit pentru

$$S_w = \sum_{k=0}^{ns} w_k$$

Pentru $i = 0, np - 1$ **execută**

$w_k = \frac{w_k}{S_w}$ /*normalizarea ponderilor w*/

sfârșit pentru

Algoritmul 5.2 calculează semi-variațele între punctele eșantion și normalizează ponderile asociate. Aceasta este o metodă utilizată în analiza spațială pentru a evalua variațiile valorilor măsurate în funcție de distanța dintre puncte. În algoritm, inițial se calculează numărul de puncte de semi-variață ns. Valorile s_k (semi-variațele) și w_k (ponderile) sunt inițializate cu 0.

Pentru fiecare pereche de puncte (i, j) se calculează distanța d dintre punctele P_i și P_j. Dacă distanța d este mai mică decât $maxdist + toler$ (o distanța maximă admisă între două puncte), se determină indicele k ca fiind $d / toler$. Se actualizează valoarea semi-variației s_k adăugând pătratul diferenței dintre valorile măsurate în punctele i și j, respectiv, v_i și v_j. Se incrementează numărul de perechi w_k. În următorul pas, are loc calculul semi-variației, iar ultimul pas este normalizarea ponderilor.

Algoritmul 5.3 *Calcularea coeficienților de semi-variației folosind metoda celor mai mici pătrate*

Parametri de intrare:

- Distanța maximă dintre punctele de influență: maxdist = 100km;
- np - numărul de puncte eșantion
- P_i – coordonatele celui de-al i-lea punct eșantion ($i = 0, np - 1$)
- v_i – valoarea măsurată în cel de-al i-lea punct eșantion ($i = 0, np - 1$)

Parametri de ieșire:

- W_k- ponderile $k = 0, ns - 1$

$np_2 = 0$
Pentru $i = 0, np\ -\ 1$ **execută**
  Pentru $k = 0, np\ -\ 1$ **execută**
    Dacă $k \neq i$ **atunci**
      $d = dist(P_i, P_j)$
      Dacă $d < maxdist$ **atunci**
        se calculează semi-varianța σ^1_{np2}
        $P2_{np2} = P_i$
        $np_2 = np_2 + 1$
      sfârșit dacă
    sfârșit dacă
  sfârșit pentru
  Pentru $k = 0, np_2\ - 1$ **execută**
    Pentru $l = 0, np_2 - 1$ **execută**
      $h = dist(P2_k, P2_l)$
      se calculează semi-varianța $\sigma^2_{k,l} = \sigma^2_{l,k}$ folosind h
      (**Algoritmul 5.2**)
    sfârșit pentru
  sfârșit pentru
  se rezolvă sistemul liniar de ecuații $\sigma^2_w W = \sigma^1$ pentru a calcula ponderile W_k($k = 0, ns - 1$)
sfârșit pentru

Algoritmul 5.3 este utilizat pentru a calcula coeficienții de semi-varianță folosind metoda celor mai mici pătrate. Aceasta este o metodă statistică utilizată în geostatistică pentru a estima variabilitatea spațială a unui set de date măsurate. În prima parte a algoritmului se inițializează variabilele necesare pentru calculul semi-variațelor. În pasul următor se parcurg toate punctele și se calculează semi-variațele pentru punctele care sunt la o distanță mai mică decât maxdist, iar punctele care îndeplinesc această condiție sunt adăugate într-o listă $P2$. Se calculează mai apoi semi-variațele pentru toate perechile de puncte din lista $P2$. Pentru calculul ponderilor W_k se construiește și se rezolvă un sistem de ecuații liniare.

Algoritmul 5.4 *Calcularea valorilor interpolate*

Parametri de intrare:

- nq - numărul de puncte în care se face estimarea
- np - numărul de puncte eșantion
- Q_i – coordonatele celui de-al i-lea punct de estimat ($i = 0, np - 1$)
- v_i – valoarea măsurată în cel de-al i-lea punct eșantion ($i = 0, np - 1$)

Parametri de ieșire:

- u_i- valorile estimate în fiecare punct de coordonate Q_i ($i = 0, nq - 1$)

Pentru $i = 0, nq - 1$ **execută**

se găsesc punctele eșantion aflate la distanțe mai mici de maxdist față de Q_i

se calculează u_i folosind ponderile din W

sfârșit pentru

Algoritmul 5.4 descrie modul de calculare a valorilor interpolate în punctele de estimare utilizând valorile măsurate în punctele eșantion și ponderile calculate cu ajutorul **Algoritmul 5.3**.

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-varianței 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-variațele 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

Există multe studii care stabilesc relația strânsă de cauzalitate dintre poluarea aerului și anumite boli respiratorii sau mortalitatea crescută (Schwartz, Particulate air pollution and chronic respiratory disease 1993) (Schwartz, Air pollution and daily mortality: a review and meta-analysis 1994). S-a observat că viețile oamenilor sunt scurtate cu o medie de aproape trei ani din cauza diferitelor surse de poluare a aerului (Lelieveld et al. 2020). Astfel, expunerea pe termen lung la poluarea aerului contribuie la cauza multor boli precum boli cardiovasculare și respiratorii, cancer pulmonar sau infecții ale tractului respirator inferior.

Chiar dacă în ultimii ani s-au făcut eforturi uriașe pentru îmbunătățirea calității aerului în zonele urbane (restructurare industrială, schimbări tehnologice și control al poluării), problema poluării aerului rămâne și trebuie monitorizată și studiată cu mare atenție pentru ca cetățenii să poată fi avertizați și protejați. Având în vedere toate acestea, este evident că, cartografierea poluării din mediul urban poate fi de mare ajutor institutelor care se ocupă de calitatea aerului pentru stabilirea zonelor cu risc ridicat de îmbolnăvire.

Câteva dintre moleculele care contribuie cel mai mult la poluarea aerului sunt:

- Monoxidul de carbon – CO. Cele mai mari surse de CO pentru aerul exterior sunt vehiculele sau utilajele pe baza de combustibili fosili. Chiar dacă este puțin probabil ca nivelurile ridicate de CO să apară în aer liber. Când se întâmplă acest lucru, poate afecta starea de sănătate, în special a persoanelor cu anumite tipuri de boli cardiace.
- Dioxidul de sulf – SO_2. Pe lângă emisiile de la arderea combustibililor fosili, alte surse de SO_2 sunt centralele electrice, industria de prelucrare a metalelor sau instalațiile de topire. Persoanele vulnerabile la concentrații mari de SO_2 sunt în special copii și vârstnici și, evident, cei cu afecțiuni preexistente. SO_2 este, de asemenea, responsabil pentru formarea de smog sau ceață.
- Dioxid de azot - NO_2. Ca și ceilalți poluanți, NO_2 este eliberat în aer din combustibilul care arde. Efectul principal pentru sănătatea oamenilor este iritarea căilor respiratorii. Funcția timpului de expunere și a concentrației de NO_2 poate provoca sau agrava boli respiratorii precum astmul. De asemenea,

atunci când interacționează cu apa atmosferică, NO_2 formează ploi acide care sunt dăunătoare pentru ecosisteme.

- Particulele de diferite dimensiuni (PM). PM este un termen general pentru particulele mici sau picăturile de lichid din atmosferă. De exemplu, PM10 sunt particulele cu un diametru mai mic de 10μm. Sursele primare de PM sunt arderea incompletă, emisiile vehiculelor, praful. Expunerea la poluarea cu particule poate cauza sau agrava boli de inimă sau plămâni. Prin urmare, acestea reprezintă și un pericol pentru sănătatea publică.

Chiar dacă orașul Brașov este situat între munți și înconjurat de regiuni mari de pădure, conform raportului Alianța Europeană pentru Sănătate Publică 2020 (de Bruyn și de Vries 2020), acest oraș se află în top 5 cele mai poluate orașe din Europa cu o concentrație foarte mare de NO_2. Astfel, este nevoie de o monitorizare și un studiu atent al evoluției poluării în aceasta regiune pentru a putea întreprinde acțiunile legale adecvate pentru creșterea calității aerului și, astfel, pentru a îmbunătăți sănătatea populației și a reduce costurile financiare și sociale. S-a utilizat metoda IDW pentru a crea hărți de poluare pentru zona urbană a Brașovului și a trage concluzii privind poluarea pentru anul 2020. De asemenea, s-a făcut o comparație a calității aerului în perioada de lock-down (majoritatea activităților economice și sociale au fost oprite) din cauza pandemiei de Covid-19 și perioada în care economia funcționa la maxim. 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. Coordonatele celor patru stații sunt:

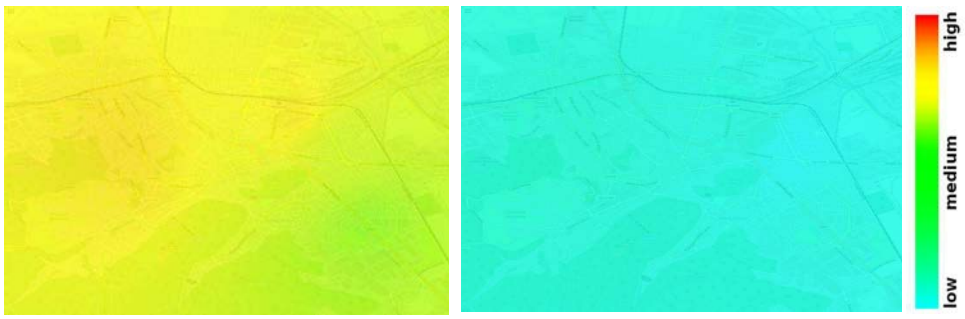
- BV-1: latitude = 45.637722, longitude = 25.630704
- BV-2: latitude = 45.652490, longitude = 25.583146
- BV-3: latitude = 45.659380, longitude = 25.616236
- BV-5: latitude = 45.651580, longitude = 25.625683

Am aplicat algoritmul IDW pentru a obține hărțile de poluare cu rezoluție de 600 x 900 (m = 600 – pe latitudine și n = 900 pe longitudine). Coordonatele geografice ale gridului considerat sunt:

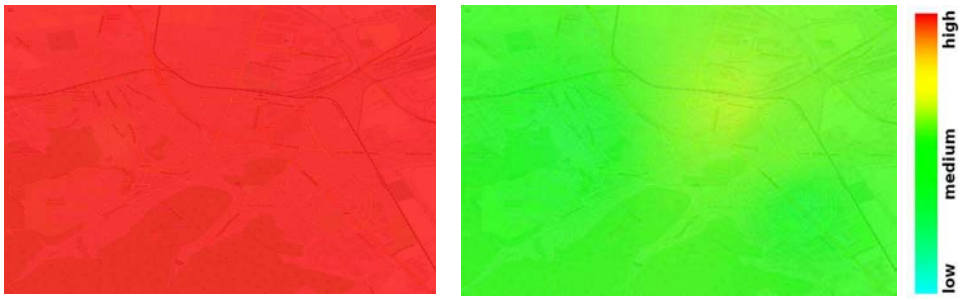
$ymin = 45.62;$
$ymax = 45.68;$
$xmin = 25.56;$
$xmax = 25.65$

Este ușor de observat că $dx = dy$ în IDW (ambele sunt egale cu 10^{-4}), astfel încât aspect ratio în sistemul de coordonate standard al WGS 84 (World Geodetic System 1984) a fost păstrat.

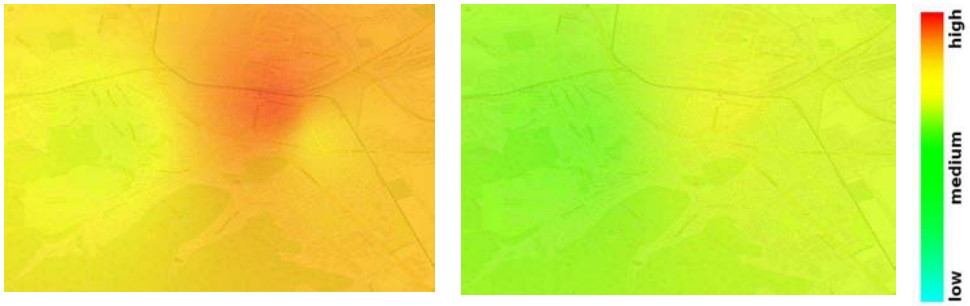
Folosind algoritmul IDW, s-au generat hărți ale concentrațiilor poluanților din oră în oră, hărți cu mediile pe 24 de ore pentru fiecare poluant, hărți cu mediile concentrațiilor pe o lună și hărți cu mediile concentrațiilor pentru fiecare zi a săptămânii pentru a vedea cum diferă poluarea in zilele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



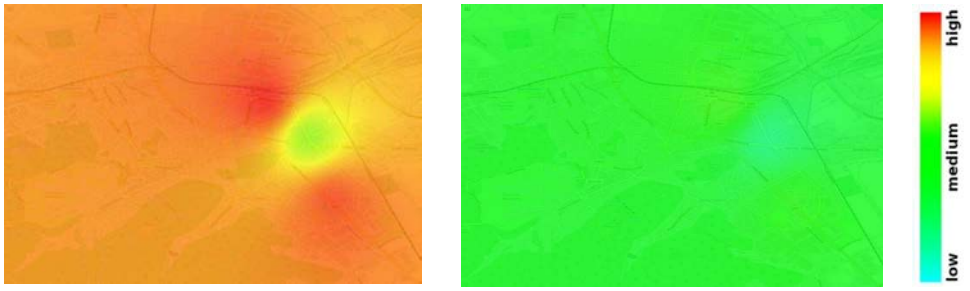
(a) CO - Ianuarie (b) CO – Mai



(c) NO_2 - Ianuarie (d) NO_2 – Mai



(e) SO_2 - Ianuarie (f) SO_2 – Mai

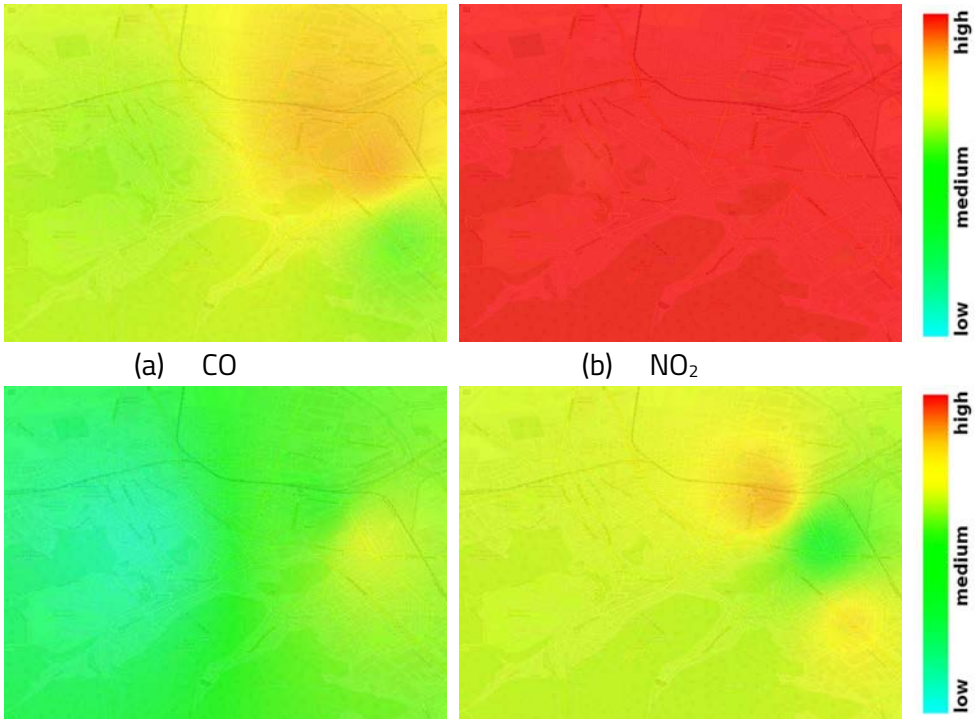


(g) PM10 - Ianuarie (h) PM10 – Mai

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

În primul rând, pentru fiecare poluant s-au creat două hărți pentru a compara concentrația medie în ianuarie (înainte de lock-down) și mai (ultima lună de lock-down) (**Figura 5.1**). Comparând cele două imagini, este evidentă îmbunătățirea semnificativă a calității aerului pentru fiecare dintre factorii poluanți datorită activității industriale reduse și a numărului scăzut de vehicule care au circulat în perioada respectivă.

În **Figura 5.2**, sunt prezentate hărțile pentru momentele de timp în care au fost înregistrate cele mai mari valori orare ale concentrațiilor fiecărui poluant. Toate valorile maxime au fost înregistrate în ianuarie, iar cea mai mare valoare a fost pentru NO_2. Astfel, valoarea maximă înregistrată pentru NO_2 a fost de $178\ \mu g\ /\ m^3$, această valoare fiind foarte apropiată de valoarea maximă admisă pentru o scurtă perioadă (**Tabel 5.1**).



(a) CO (b) NO_2

(c) SO_2 (d) PM10

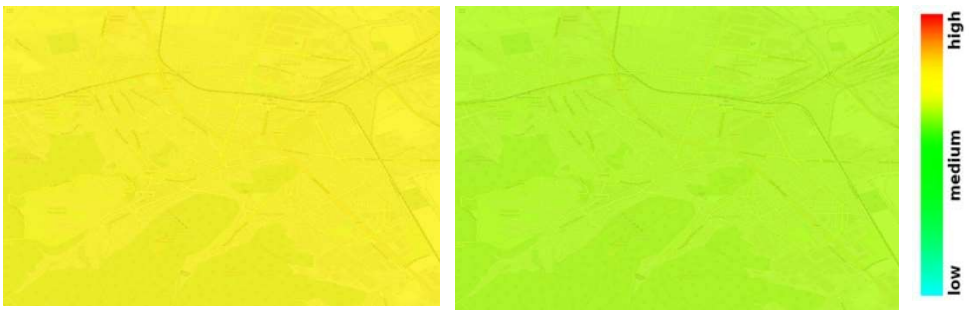
Figura 5.2 *Hărțile cu concentrația maximă pentru fiecare poluant*

Tabel 5.1 *Valorile maxime admise pentru concentrațiile principalilor poluanți* **(Rețeaua Națională de Monitorizare a Calității Aerului 2021)**

Poluant	Concentrația maximă admisă pentru perioadă scurtă (mg / m^3)	Concentrația maximă admisă pentru perioadă lungă (mg / m^3)
CO	10	-
NO_2	0.20	0.04
SO_2	0.35	0.125
PM10	0.05	0.04

După cum am menționat anterior, Brașovul este unul dintre primele 5 orașe din Europa în ceea ce privește concentrația de NO_2. În **Figura 5.3**, este prezentată o comparație între valorile medii ale NO_2, pentru întreaga perioadă analizată, în două zile ale săptămânii și, în **Figura 5.4**, este prezentată diagrama cu evoluția valorii medii a concentrației de NO_2 pentru toate zilele săptămânii. După cum era

de așteptat, valoarea zilei de duminică este semnificativ mai mică decât valorile zilelor lucrătoare.



(a) Joi (b) Duminica

Figura 5.3 *Hărțile cu concentrația medie de NO_2 în două zile ale săptămânii*

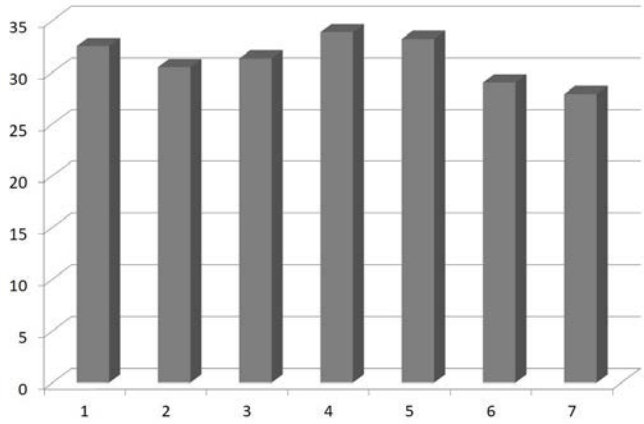
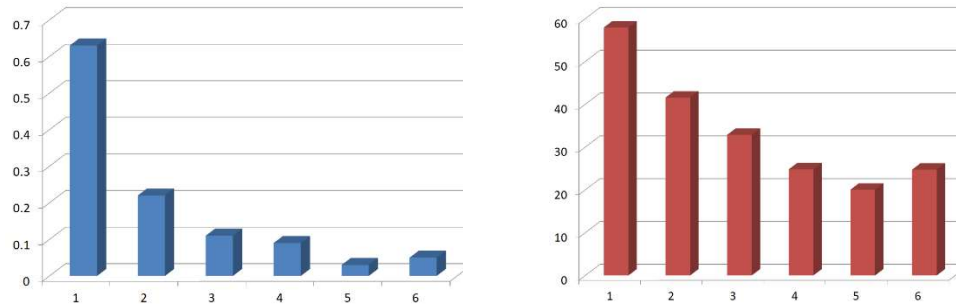


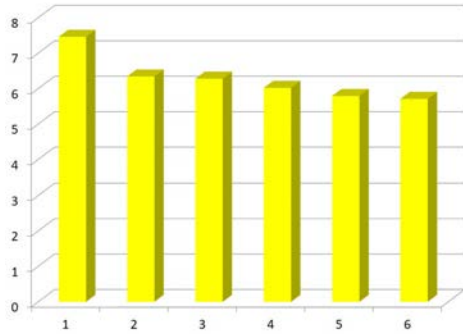
Figura 5.4 *Concentrația medie de NO_2 ($\mu g/m^3$) pentru fiecare zi ale săptămânii pentru întreaga perioadă studiată*

În **Figura 5.5**, este prezentat graficul evoluției valorii medii a NO_2 pentru prima jumătate a anului 2020. Se observă că valorile medii lunare ale concentrației fiecărui poluant au scăzut în perioada de izolare.

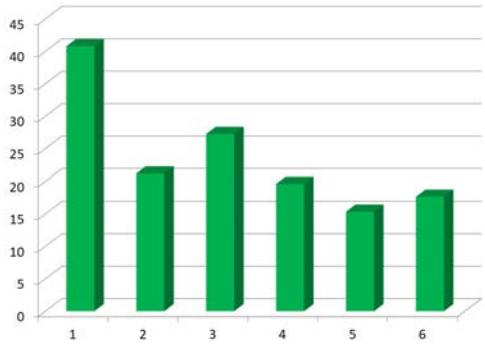


(a) CO

(b) NO_2



(c) SO_2



(d) PM10

Figura 5.5 *Evoluția mediilor lunare ale concentrațiilor poluanților în prima jumătate a anului 2020. Unitatea de măsura pentru CO_2 este $^{mg}/_{m^3}$, iar pentru NO_2, SO_2 și PM 10 este $^{\mu g}/_{m^3}$*

Rezultatele experimentale prezentate în **Tabel 5.2** 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 folosită 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 5.2 *Creșterea 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 geomagnetice sunt informațiile adunate despre câmpul magnetic al Pământului. Datele geomagnetice pot fi colectate folosind diferite metode cum ar fi stațiile fixe dotate cu sonde magnetice sau măsurătorile magnetice efectuate de aeronave, nave (Blakely 1996) (Campbell 1996). Hărțile geomagnetice sunt reprezentări grafice ale acestor date care arată variații ale intensității și direcției câmpului magnetic în diferite locații de pe suprafața Pământului (Thébaul et al. 2015).

Datele și hărțile geomagnetice sunt utilizate într-o gamă largă de aplicații științifice, industriale și comerciale, de la înțelegerea structurii Pământului până la identificarea potențialelor resurse minerale și energetice. Astfel, datele geomagnetice sunt esențiale pentru înțelegerea structurii și dinamicii interiorului Pământului (McElhinny and McFadden 2000). Câmpul magnetic este generat de mișcarea fierului topit în miezul exterior al Pământului, iar variațiile câmpului pot oferi perspective asupra structurii și comportamentului regiunii (Kivelson și Russell 2017). De exemplu, schimbările în puterea și direcția câmpului magnetic pot indica prezența structurilor geologice, cum ar fi falii, camere magmatice și depozite minerale (Telford, Geldart și Sheriff 1990).

Datele geomagnetice sunt folosite, de asemenea, pentru a studia atmosfera superioară a Pământului și interacțiunea acesteia cu vântul solar. Câmpul magnetic acționează ca un scut, deviind particulele de înaltă energie de la soare și protejând atmosfera Pământului de eroziune (Parker 1990) . Cu toate acestea, câmpul magnetic nu este uniform, iar variațiile în puterea și direcția acestuia pot duce la formarea unor regiuni cunoscute sub numele de centuri de radiație Van Allen, unde particulele încărcate sunt prinse și pot provoca daune navelor spațiale și sateliților.

Datele geomagnetice sunt folosite de asemenea în navigație și topografie. Câmpul magnetic oferă o referință pentru busole și poate fi folosit pentru a determina orientarea și poziția obiectelor de pe suprafața Pământului. Aceste

informații sunt utilizate într-o gamă largă de aplicații, de la cartografiere și topografie la navigație și explorare minerală (Campbell 1996).

Hărțile geomagnetice sunt reprezentări grafice ale câmpului magnetic al Pământului. Ele arată variații în puterea și direcția câmpului în diferite locații de pe suprafața Pământului și pot fi utilizate pentru a identifica structuri geologice, zăcăminte minerale și alte caracteristici (Thébault et al. 2015).

Hărțile geomagnetice sunt utilizate într-o gamă largă de aplicații, inclusiv explorarea minerală, unde variațiile câmpului magnetic pot indica prezența zăcămintelor minerale. De exemplu, Bazinul Sudbury din Canada este o structură geologică mare care conține un depozit bogat de nichel, cupru și alte minerale (Chave and Jones 2012). Bazinul este marcat de o anomalie magnetică mare, care a fost identificată prin sondaje magnetice și a ajutat la ghidarea explorării și dezvoltării zăcământului.

Hărțile geomagnetice sunt folosite și în monitorizarea mediului, unde modificările câmpului magnetic pot indica prezența poluanților sau a altor contaminanți. De exemplu, măsurătorile magnetice au fost folosite pentru a cartografia amploarea deversărilor de petrol și a altor dezastre de mediu, oferind informații valoroase pentru eforturile de curățare și atenuare.

Hărțile geomagnetice sunt, de asemenea, folosite în cercetarea geofizică, unde pot oferi perspective asupra structurii și comportamentului interiorului Pământului. De exemplu, anomaliile magnetice din scoarța oceanică au fost folosite pentru a studia procesul de răspândire a fundului mării și formarea de noi cruste oceanice.

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 geomagne-tice

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.

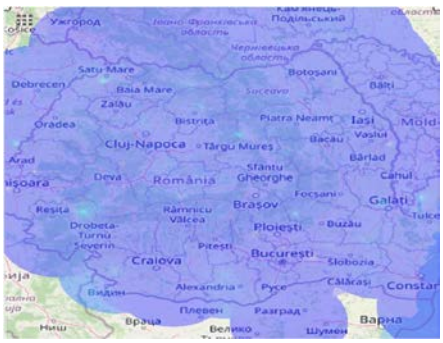


Figura 5.6 *Harta geomagnetică obținută prin IDW*

Figura 5.7 *Harta geomagnetică obținută prin interpolare kriging*

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 5.6** și **Figura 5.7** sunt prezentate hărți geomagnetice cu rezoluție de 1 km pentru regiunea României obținute prin IDW și, respectiv, interpolarea kriging.

Acuratețea rezultatelor este prezentată în **Tabel 5.3**. Diferența medie între valoarea interpolată și cea reală, abaterea standard, valoarea mediană și diferența maximă sunt prezentate pentru fiecare rezoluție studiată, pentru ambele metode de interpolare. Eroarea a fost obținută eliminând câte un punct și calculând diferența dintre valoarea interpolată și valoarea reală măsurată în fiecare punct specific. Rezultatele au o mai bună acuratețe 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 5.8**, pot fi observate valorile mai mici ale erorii medii a câmpului geomagnetic pentru metoda kriging.

Tabel 5.3 *Acuratețea metodelor de interpolare*

Metode de interpolare	*Rezoluția gridului*	*Eroarea mediana*	*Eroarea maxima*	*Eroarea medie*	*Deviația standard*
IDW	400 x 400	4.895	9.21	3.600	0.275
	800 x 800	4.550	9.271	3.137	0.289
	1200 x 1200	4.495	9.268	3.065	0.254
	1600 x 1600	4.476	9.274	2.941	0.247
Kriging	400 x 400	3.687	7.21	3.102	0.312
	800 x 800	3.551	7.02	2.542	0.271
	1200 x 1200	3.215	6.82	2.14	0.252
	1600 x 1600	2.871	6.01	1.92	0.248

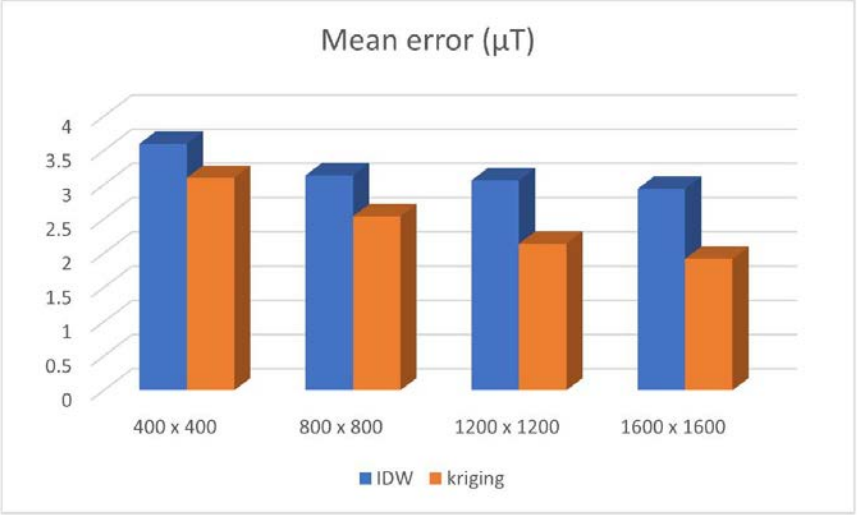


Figura 5.8 *Comparație între eroarea medie pentru câmpul geomagnetic obținut prin IDW și Kriging*

Cu alte cuvinte, comparând rezultatele, se poate 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.

În **Tabel 5.4** sunt afișați timpii de execuție pe CPU și GPU și speed-up-ul pentru algoritmii implementați CUDA. După cum s-a putut observa, calculul complex din metoda kriging duce la un timp de execuție crescut pentru toate rezoluțiile, în comparație cu IDW **Figura 5.9**. 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 5.10**, 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.

Tabel 5.4 *Timpii de execuție și accelerările pe GPU ale algoritmilor IDW și kriging*

Metode de interpolare	Rezoluția gridului	Timpul de execuție pe CPU (s)	Timpul de execuție pe GPU (s)	Speed-up
IDW	400 x 400	13.7	0.355	38.6
	800 x 800	54.1	0.7204	74.4
	1200 x 1200	121.7	1.265	96.2
	1600 x 1600	212.8	2.044	104.1
Kriging	400 x 400	49.4	11.9	4.15
	800 x 800	190.5	33.0	5.77
	1200 x 1200	440.9	53.0	8.32
	1600 x 1600	757.1	72.4	10.45

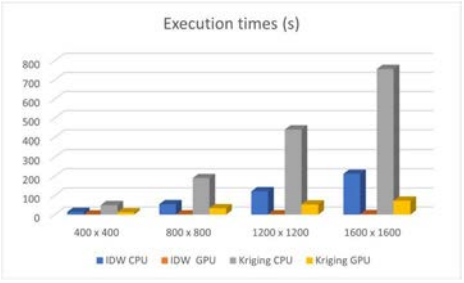


Figura 5.9 *Timpii de execuție pentru IDW și Kriging pe CPU și GPU*

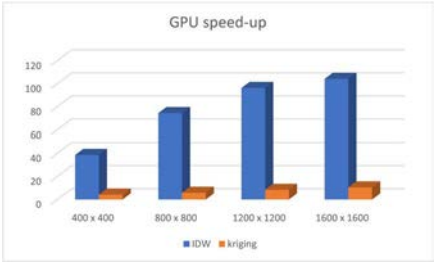


Figura 5.10 *Speed-up-ul pentru IDW și Kriging*

Astfel, din rezultatele obținute, 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 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 două 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.

Concluzii

CUDA este o tehnologie populară cu aplicații recente numeroase în domenii variate. Cu ajutorul CUDA, programatorii pot exploata puterea de procesare a GPU-urilor pentru a accelera rezolvarea problemelor complexe și pentru a obține performanță superioară într-o varietate de domenii. Aceasta arhitectura oferă o abordare flexibilă pentru programarea pe GPU, permițând dezvoltatorilor să scrie cod într-un mod familiar, asemănător cu limbajul C. Totodată, această abordare facilitează tranziția pentru programatorii care au experiență anterioară în dezvoltarea de software pe procesoare tradiționale.

CUDA este optimizată pentru execuția eficientă a operațiilor paralele pe GPU. Prin intermediul modelului de programare CUDA, dezvoltatorii pot exploata puterea masivă de procesare paralelă a GPU-urilor, accelerând astfel aplicațiile care beneficiază de această abordare.

CUDA nu este limitată doar la domeniul procesării grafice, ci este utilizată într-o varietate de domenii, cum ar fi deep - learning, simulări științifice, analiza datelor, și altele. Această versatilitate face din CUDA o alegere potrivită pentru dezvoltatori din diverse industrii.

CUDA beneficiază de o comunitate mare și activă de dezvoltatori și cercetători. Acest lucru facilitează schimbul de cunoștințe și împărtășirea de soluții în cadrul comunității, contribuind la evoluția continuă a arhitecturii.

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 ale autoarei 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 lucrarea (Deaconu și **Spridon** 2021).

În **Capitolul 3**, se introduce și se rezolvă 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 sursă la un alt nod dat t numit s 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. 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** am obținut hărți georeferențiate cu ajutorul metodelor de interpolare bidimensionale și pornind de la valori măsurate în puncte discrete, într-o anumită zonă geografică care au fost implementate folosind CUDA. 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).

Rezumând, programarea CUDA în teoria grafurilor deschide o serie de perspective interesante și utile pentru rezolvarea problemelor complexe asociate cu această ramură a matematicii discrete. De asemenea utilizarea metodelor GPU pot fi folosite pentru obținerea rapide de hărți bidimensionale de rezoluție și acuratețe ridicate.

Bibliografie

Aguilar-Sánchez, R., I.F. Herrera-González, J.A. Méndez-Bermúdez, și J.M. Sigarreta. „Computational Properties of General Indices." *Symmetry* 12 (2020): 1341.

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

Albert, R., și A.L. Barabási. „Statistical mechanics of complex networks." *Reviews of Modern Physics*, 2002: 47-97.

Almasan, P., J. Suárez-Varela, K. Rusek, P. Barlet-Ros, și A Cabellos-Aparicio. „Deep reinforcement learning meets graph neural networks: Exploring a routing optimization use case." *Computer Communications*, 2022: 184-194.

Apanasevich, L., Y. Kale, H. Sharma, și A. M. Sokovic. *A Comparison of the Performance of the Molecular Dynamics Simulation Package GROMACS Implemented in the SYCL and CUDA Programming Models.* 09 2023. https://ieee-hpec.org/wp-content/uploads/2023/09/150.pdf.

Athanasopoulos, A., A. Dimou, V. Mezaris, și I. Kompatsiaris. „GPU acceleration for support vector machines." *12th Inter. Workshop on Image Analysis for Multimedia Interactive Services .* Delft, 2011.

Auer, B F, și R Bisseling. „Graph coarsening and clustering on the GPU." *Graph Partitioning and Graph Clustering*, 2012.

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.

Bali, M., și S.P. Anandaraj. „Biomolecular Event Extraction using Natural Language Processing." *International journal of electrical and computer engineering systems* 14, nr. 5 (2023): 601-612.

Barrionuevo, M., M. Lopresti, N. C. Miranda, și M. F. Piccoli. „Solving a big-data problem with GPU: the network traffic analysis." *Journal of Computer Science and Technology* 15, nr. 01 (2015): 30-39.

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

Belloch, J. A., B. Bank, L. Savioja, A. Gonzalez, și V. Välimäki. „Multi-channel IIR filtering of audio signals using a GPU." *IEEE International Conference on Acoustics, Speech and Signal Processing.* Florence: IEEE, 2014. 6692-6696.

Bernaschi, M., M. Fatica, Melchionna, Succi, S. S., și E. Kaxiras. „A flexible high-performance Lattice Boltzmann GPU code for the simulations of fluid flows in complex geometries." *Concurrency and computation: practice and experience* 22, nr. 1 (2010): 1-14.

Bharadiya, J.P. „A comparative study of business intelligence and artificial intelligence with big data analytics." *American Journal of Artificial Intelligence* 7, nr. 1 (2023): 24.

Blakely, R. J. *Potential theory in gravity and magnetic applications.* Cambridge university press, 1996.

Bolz, J, I Farmer, E Grinspun, și P Schröder. „Sparse matrix solvers on the GPU: conjugate gradients and multigrid." *ACM Transactions on Graphics* 22, nr. 3 (2003): 917-924.

Brand, J., și alții. „Bipartite Matching in Nearly-linear." *IEEE 61st Annual Symposium on Foundations of Computer Science.* Durham, NC, USA: IEEE, 2020. 919–930.

Burrough, P.A. *Principles of geographical information systems for land resources assessment.* Oxford, Clarendon Press, 1986.

Cai, Y. „Research on the Construction of Financial Computing Model Based on BSDE Algorithm." *Journal of Information & Knowledge Management*, 2023: 2350029.

Campbell, W. H. *Introduction to geomagnetic fields.* Cambridge university press, 1996.

Chave, A. D., și A. G. Jones. *The magnetotelluric method.* Cambridge university press, 2012.

Chen, C., K. Li, A. Ouyang, Z. Zeng, și K. Li. „GFlink: An in-memory computing architecture on heterogeneous CPU-GPU clusters for big data." *IEEE Transactions on Parallel and Distributed Systems*, 2018: 1275-1288.

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.

Cressie, N. *Statistics for spatial data (revised edition).* New York: Wiley, 1993.

Cugola, G., și A. Margara. „Low latency complex event processing on parallel hardware." *Journal of Parallel and Distributed Computing* 72, nr. 2 (2012): 205-218.

Cuomo, S., V. De Angelis, G. Farina, L. Marcellino, și G. Toraldo. „A GPU-accelerated parallel K-means algorithm." *Computers & Electrical Engineering*, 2019: 262-274.

D'Agostino, D., I. Merelli, M. Aldinucci, și D. Cesini. „Hardware and software solutions for energy-efficient computing in scientific programming." *Scientific Programming*, 2021: 1-9.

Dantzig, G. B. „Applications of the simplex method to a transportation problem." În *Activity analysis and production and allocation*, de T. C. Koopmans, 359–373. New York: Wiley, 1951.

—. *Linear programming and extensions.* NJ: Princeton University Press, 1962.

de Bruyn, S., și J. de Vries. *https://cleanair4health.eu/.* 2020. https://cleanair4health.eu/wp-content/uploads/sites/2/2020/10/final-health-costs-of-air-pollution-in-european-cities-and-the-linkage-with-transport-c.pdf.

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.

Du, P., R. Weber, P. Luszczek, S. Tomov, G. Peterson, și J. Dongarra. „From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming." *Parallel Computing* 38 (8) (2012): 391-407.

Ducruet, C., și I. Lugo. „Structure and dynamics of transportation networks: models, methods and applications." *The SAGE handbook of transport studies*, 2013: 347-364.

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

Erdõs, P., și A. Rényi. „On Random Graphs I." *Publicationes Mathematicae Debrecen*, 1959: 290-297.

Feitoza Santos, A., de Lacerda Fernandes, S. F., P. Gomes Lopes Júnior, D. Fawzi Hadj Sadok, și G. Szabo. „Multi-gigabit traffic identification on GPU." *High performance and programmable networking.* New York City: ACM, 2013. 39-44.

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

Franke, R. „Thin plate spline with tension." *Computer Aided Geometrical Design*, 1985: 87-95.

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.

Fu, Z., J. Zhou, W. Xu, C. Guo, și Q. Wu. „GPU and VPU Enabled Virtual Mobile Infrastructure for 3D Image Rendering and Its Application in Telemedicine." *IEEE Internet of Things Journal*, 2023.

Fung, J., și S. Mann. „OpenVIDIA: parallel GPU computer vision." *Proceedings of the 13th ACM International Conference on Multimedia.* Singapore: ACM, 2005.

Gale, T., M. Zaharia, C. Young, și E. Elsen. „Sparse GPU Kernels for Deep Learning." *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis.* Atlanta, 2020. 1-14.

Gao, S., J. Cao, și J. Yang. „Parallel implementation of kriging interpolation based on CUDA." *Journal of Computational and Applied Mathematics* 370 (2020): 112665.

Gao, Y., Y.P. Liu, și R. Peng. „Fully Dynamic Electrical Flows: Sparse Maxflow Faster Than Goldberg-Rao." *arXiv*, 2021.

Goodarzi, B, M Burtscher, și D Goswami. „Parallel Graph Partitioning on a CPU-GPU Architecture." *2016 IEEE International Parallel and Distributed Processing Symposium Workshops.* Chicago: IEEE, 2016. 58-66.

Guo, Y., și alții. „Rapid detection of non-normal teeth on dental X-ray images using improved Mask R-CNN with attention mechanism." *International Journal of Computer Assisted Radiology and Surgery*, 2024: 1-12.

Hancock, J. T., T. M. Khoshgoftaar, și J. M. Johnson. „Evaluating classifier performance with highly imbalanced Big Data." *Journal of Big Data* 10, nr. 1 (2023): 42.

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

Haseeb, M., N. Ding, J. Deslippe, și M. Awan. „Evaluating Performance and Portability of a core bioinformatics kernel on multiple vendor GPUs." *International Workshop on Performance, Portability and Productivity in HPC.* St. Louis: IEEE, 2021. 66-78.

Hoshino, T., N. Maruyama, S. Matsuoka, și R. Takaki. „CUDA vs OpenACC: Performance case studies with kernel benchmarks and a memory-bound CFD application." *13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing.* Delft: IEEE, 2013. 136-143.

Howes, B. *OpenCL Programming by Example.* Packt Publishing, 2013.

Hu, L., L. Zou, și Y. Liu. „Accelerating Triangle Counting on GPU." *International Conference on Management of Data.* New York: Association for Computing Machinery, 2021.

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

i Cancho, R. F., C. Janssen, și R. V. Solé. „Topology of technology graphs: Small world patterns in electronic circuits." *Physical Review E*, 2001: 046119.

Jewell, W. S. „Optimal flow through networks with gains." *Operations Research*, 1962: 476–499.

Jewell, W. S. „Optimal flow through networks. Technical Report 8." *Operations Research Center*, 1958.

Ji, X., H. Wu, și J. Wang. „A CUDA-based framework for IDW and Kriging interpolation." *International Journal of Geographical Information Science, 34(8), 1572-1587*, 2020: 1572-1587.

Jiang, H.,, Y., Qiao Chen, T.-H. Z. Weng, și K.-C. Li. „Scaling up MapReduce-based Big Data Processing on Multi-GPU systems." *Cluster Computing* 18, nr. 1 (2015): 369-383.

Jiang, W. „Graph-based deep learning for communication networks: A survey." *Computer Communications*, 2022: 40-54.

Jog, A., Kayiran, O., Kesten, T., Pattnaik, A., Bolotin, E., Chatterjee, N., S.W. Keckler, M.T. Kandemir, și C. R. Das. „Anatomy of gpu memory system for multi-application execution." *Proceedings of the 2015 International Symposium on Memory Systems.* Washington DC: ACM, 2015. 223-234.

Journel, A. G., și C. J. Huijbregts. *Mining geostatistics.* London: Academic Press, 1978.

Kang, S., J. Nke, și Rees B. „Analyzing Multi-trillion Edge Graphs on Large GPU Clusters: A Case Study with PageRank." *IEEE Conference on High Performance Extreme Computing (HPEC).* Waltham: IEEE, 2022.

Karakostas, G. „Faster approximation schemes for fractional multicommodity flow problems." *13th Annual ACM-SIAM Symposium on Discrete Algorithms,.* San Francisco, CA, USA: ACM, 2002. 166–173.

Karimi, K., N. G. Dickson, și F. Hamze. „A performance comparison of CUDA and OpenCL." *arXiv preprint arXiv:1005.2581*, 2010.

Karpinsky, N., și S. Zhang. „Holovideo: Real-time 3D range video encoding and decoding on GPU." *Optics and Lasers in Engineering*, 2012: 280-286.

Katz, G.J., și J.T. Kider. „All-Pairs Shortest-Paths for Large Graphs on the GPU." *23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware.* 2008. 47-55.

Kersten, T. P., și D. Stallmann. „Automatic texture mapping of architectural and archaeological 3d models." *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* 39 (2012): 273-278.

Khalilov, M., și A. Timoveev. „Performance analysis of CUDA, OpenACC and OpenMP programming models on TESLA V100 GPU." *Journal of Physics: Conference Series.* Moscow: IOP Publishing, 2020. 012056.

Khani, M., V. Sivaraman, și M. Alizadeh. „Efficient video compression via content-adaptive super-resolution." *International Conference on Computer Vision.* Montreal, 2021. 4521-4530.

Kirk, D., și W. Hwu. *Programming Massively Parallel Processors.* Morgan Kaufmann, 2016.

Kivelson, M. G., și C. T. Russell. *Introduction to space physics.* Cambridge university press, 2017.

Klukovich, E, M H Gunes, L. Barford, și F C Harris. „Accelerating BFS shortest paths calculations using CUDA for Internet topology measurements." *2016 International Conference on High Performance Computing & Simulation (HPCS).* Innsbruck: IEEE, 2016. 66-73.

Klukovich, E., M. Hadi Gunes, L. Barford, și F. C. Harris. „Accelerating BFS shortest paths calculations using CUDA for Internet topology measurements." *2016 International Conference on High Performance Computing & Simulation.* Innsbruck: IEEE, 2016. 66-73.

Krüger, J., și R. Westermann. „Linear algebra operators for GPU implementation of numerical algorithms." *ACM Transactions on Graphics* 22, nr. 3 (2003): 908-916.

Kumar, S., și K.K. Mohbey. „A review on big data based parallel and distributed approaches of pattern mining." *Journal of King Saud University - Computer and Information Sciences* (Science Direct) 34, nr. 5 (2022): 1639-1662.

Lee, W. B., și A. G. Constantinides. „Computational Results for a Quantum Computing Application in Real-Life Finance." *414-423.* Bellevue, Washington: IEEE, 2023. IEEE International Conference on Quantum Computing and Engineering.

Lee, W. K., K. Jang, G. Song, H. Kim, S. O. Hwang, și H. Seo. „Efficient implementation of lightweight hash functions on gpu and quantum computers for iot applications." *IEEE Access*, 2022: 59661-59674.

Lelieveld, J., A. Pozzer, U. Poschl, M. Fnais, A Haines, și T. Munzel. „Loss of life expectancy from air pollution compared to other risk factors: a worldwide perspective." *Cardiovascular Research*, 2020: 1910-1917.

Li, T., M. Bolic, și P. M. Djuric. „Resampling methods for particle filtering: classification, implementation, and strategies." *IEEE Signal processing magazine* 32, nr. 3 (2015): 70-86.

Li, Z. „Geospatial big data handling with high performance computing: Current approaches and future directions." *Performance Computing for Geospatial Applications*, 2020: 53-76.

Liu, J., și L. Guo. „Implementation of Neural Network Backpropagation in CUDA." *Intelligence Computation and Evolutionary Computation. Advances in Intelligent Systems and Computing.* Berlin: Springer, 2013.

Liu, Q., C. Hu, și Y. Tian. „Parallel implementation of IDW interpolation algorithm based on CUDA." *Journal of Applied Geophysics* 189 (2021): 104332.

Liu, T., și alții. „BGL: GPU-Efficient GNN Training by Optimizing Graph Data I/O and Preprocessing." *20th USENIX Symposium on.* Boston: USENIX Association, 2023. 103-118.

Mahmoudi, M., și A. Boloori. „Network Flow Applications, in Graph Theory for Operations Research and Management: Applications in." 246–256. PA, USA: IGI Global: Hershey, 2013.

Mandal, S., K. E. Cholachgudda, L. V. Chamakura, R. C. Biradar, și G. D. Devanagavi. „etVision: A Low-Cost Real-Time Depth Estimation System using Jetson Computing Platform." *IEEE International Conference on Electronics, Computing and Communication Technologies.* Bangalore: IEEE, 2023. 1-6.

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.

McElhinny, M. W., și P. L. McFadden. *Paleomagnetism: Continents and oceans.* Academic Press, 2000.

Michalakes, J., și M. Vachharajani. „GPU acceleration of numerical weather prediction." *IEEE International Symposium on Parallel and Distributed Processing.* Miami, FL: IEEE, 2008. 1-7.

Micheli, A. „Neural network for graphs: A contextual constructive approach." *IEEE Transactions on Neural Networks*, 2009: 498-511.

Nage, S., și G. Potdar. „A Survey on Graph Partitioning Techniques." *International Journal of Science and Research* 4, nr. 9 (2015): 1656-1659.

Nejedly, P., F. Plesinger, J. Halamek, și P. Jurak. „Cuda Filters: A Signal Plant library for GPU-accelerated FFT and FIR filtering." *Software: Practice and Experience* 48, nr. 1 (2018): 3-9.

Nettleton, D. F. „Data mining of social networks represented as graphs." *Computer Science Review*, 2013: 1-34.

NVIDIA. *NVIDIA: World Leader in Artificial Intelligence Computing.* 2019. https://www.nvidia.com/en-us/.

O'Flaherty, Kate. *Nvidia, Cray, PGI, and CAPS launch 'OpenACC' programming standard for parallel computing.* 14 11 2011. http://www.theinquirer.net/inquirer/news/2124878/nvidia-cray-pgi-caps-launch-openacc-programming-standard-parallel-computing.

Oliver, M.A., și R. Webster. „A tutorial guide to geostatistics: Computing and modelling variograms and kriging." *Catena*, 2014: 56-69.

Orlin, J.B. „Max flows in O(nm) time, or better." *e 45th annual ACM symposium on Symposium on Theory of.* Palo Alto, CA, USA, 2013. 765–774.

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.

Pang, B., E. Nijkamp, și Y. N. Wu. „Deep Learning With TensorFlow: A Review." *Journal of Educational and Behavioral Statistics* 45, nr. 2 (2020): 227-248.

Parker, R. L. *Geophysical inverse theory and regularization problems.* Academic Press, 1990.

Penrose, M. *Random Geometric Graphs.* Oxford: Oxford University Press, 2003.

Prabhu, P., și alții. „A survey of the practice of computational science." *State of the practice reports*, 2011: 1-12.

Prakash, K.B., și G.R. Kanagachidambaresan. „Programming with TensorFlow. EAI/Springer Innovations in Communication and Computing." În *PyTorch*,

de S., Prakash, K.B., Kanagachidambaresan, G.R. Imambi, 87–104. Springer, 2021.

Rathore, M.M., H. Son, A. Awais, Anand P., și J. Gwanggil. „Real-Time Big Data Stream Processing Using GPU with Spark Over Hadoop Ecosystem." *International Journal of Parallel Programming* 46 (2018): 630–646.

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

„S3 Video Boards." *InfoWorld. 14 (20): 62. May 18, 1992.* 14 (20), nr. 62 (May 1992).

Sanders, J., și E. Kandrot. *CUDA by example.* Addison-Wesley, 2010.

Savioja, L., V. Välimäki, și J. O. Smith. „Audio signal processing using graphics processing units." *Journal of the Audio Engineering Society* 59 (2011): 3-19.

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

Schwartz, J. „Air pollution and daily mortality: a review and meta-analysis." *Environmental Research*, 1994: 36-52.

Schwartz, J. „Particulate air pollution and chronic respiratory disease." *Environmental Research*, 1993: 7-13.

Shevenell, M. J., J. L. Shumaker, A. H. Edwards, și R. E. Pino. „Memristors and the future of Cyber Security Hardware." *Network Science and Cybersecurity*, 2014: 273-285.

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.

Taylor-Weiner, A., și alții. „Scaling computational genomics to millions of individuals with GPUs." *Genome Biology* 20 (2019): 228.

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

Telford, W. M., L. P. Geldart, și R. E. Sheriff. *Applied geophysics.* Cambridge university press, 1990.

Thébault, E., C. C. Finlay, C. D. Beggan, și et al. „International geomagnetic reference field: the 12th generation." *Earth, Planets and Space* 67 (2015).

Van Essen, B., C. Macaraeg, M. Gokhale, și R. Prenger. „Accelerating a random forest classifier: Multi-core, GP-GPU, or FPGA?" *IEEE 20th International Symposium on Field-Programmable Custom Computing Machines.* IEEE, 2012. 232-239.

Vanderbauwhede, W., și T. Takemi. „An analysis of the feasibility and benefits of GPU/multicore acceleration of the Weather Research and Forecasting model." *Concurrency and Computation: Practice and Experience*, 2016: 2052-2072.

Viola, F., G. Del Corso, R. De Paulis, și R. Verzicco. „GPU accelerated digital twins of the human heart open new routes for cardiovascular research." *Scientific reports*, 2023: 8230.

Wagner, A., și D. A. Fell. „The small world inside large metabolic networks." *Proceedings of the Royal Society of London. Series B: Biological Sciences.* 2001. 1803-1810.

Wang, C., și X Chu. „GPU Accelerated Keccak (SHA3) Algorithm." *Computer Science*, 2019.

Wang, F, D Dong, și B Yuan. „Graph-Based Substructure Pattern Mining Using CUDA Dynamic Parallelism." *International Conference on Intelligent Data Engineering and Automated Learning.* Berlin: Springer, 2013. 342–349.

Wang, P., T. Abel, și R Kaehler. „Adaptive mesh fluid simulations on GPU." *New Astronomy* 15, nr. 7 (2010): 581-589.

Waxman, B.M. „Routing of multipoint connections." *IEEE Journal on Selected Areas in Communications*, 1988: 1617–1622.

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

Weiss, A., și A. Elsherbeni. „Computational performance of MATLAB and python for electromagnetic applications." *International Applied Computational Electromagnetics Society Symposium.* Monterey, California: IEEE, 2020. 1-2.

Wu, Z., J. Sun, Y. Zhang, Z. Wei, și J. Chanussot. „Recent Developments in Parallel and Distributed Computing for Remotely Sensed Big Data Processing." *Proceedings of the IEEE.* 2021. 1282-1305.

Wu, Z., S. Pan, F. Chen, G. Long, C. Zhang, și P.S. Yu. „A Comprehensive Survey on Graph Neural Networks." *IEEE TRANSACTIONS ON NEURAL NETWORKS AND LEARNING SYSTEMS* 32, nr. 1 (2021): 4-24.

Yang, C.l, A. Buluç, și J.D. Owens. „GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU." *ACM Transactions on Mathematical Software* 48, nr. 1 (2022): 1-51.

Yongwen, H., Z. Xiao, L. Jing, L. Binyuan, și M. Chao. „An Efficient Algorithm for Solving Minimum Cost Flow Problem with Complementarity Slack Conditions." *Mathematical Problems in Engineering*, 2020.

Zhang, G. P. „A neural network ensemble method with jittered training data for time series forecasting." *Information Sciences*, 2007: 5329-5346.

Zhang, J., S. You, și L. Gruenwald. „ Large-scale spatial data processing on GPUs and GPU-accelerated clusters." *Sigspatial Special*, 2015: 27-34.

Zhang, Q., H. Liu, și F Bu. „High performance of a GPU-accelerated variant calling tool in genome data analysis." *bioRxiv*, 2021.

Zhang, Z., și J. Li. „A Review of Artificial Intelligence in Embedded Systems." *Micromachines*, 2023: 897.