Universitatea Națională de Știință și Tehnologie POLITEHNICA din București

Facultatea de Electronică, Telecomunicații și Tehnologia Informației

Proiect de semestru

Filtrarea zgomotului Speckle din imaginile satelitare SAR

Student

Alex-Costin Țițu

Anul 2024

Cuprins

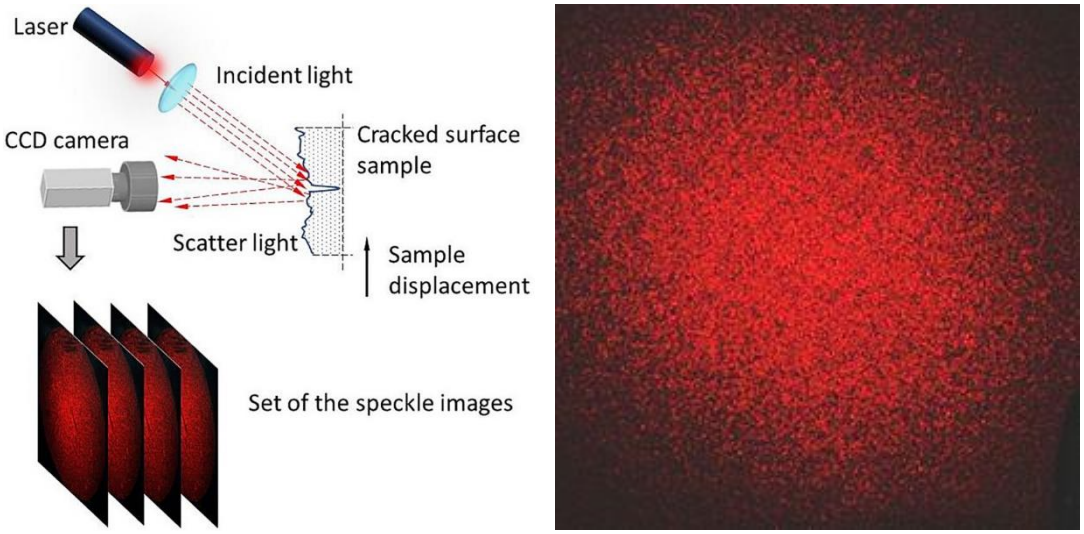
Lista Figurilor

Lista Tabelelor

1. Fundamente Teoretice

1.1. Zgomotul Speckle

Zgomotul speckle (en. granular noise), cunoscut și sub numele de model speckle sau interferență speckle, desemnează structura granulară observată în lumină coerentă, rezultând din interferență aleatorie. Speckle-ul nu este un zgomot extern, ci o fluctuație inerentă în reflexiile difuze, deoarece dispersorii (en. scatterers), numiți și surse, nu sunt identici pentru fiecare celulă, iar unda de iluminare coerentă este foarte sensibilă la variațiile mici în schimbările de fază. Modelele speckle apar atunci când lumina coerentă este aleatorizată. Cel mai simplu caz de astfel de aleatorizare este atunci când lumina se reflectă de pe o suprafață optică rugoasă, ca în figura de mai jos.[1], [2], [3]



Figură 1.1. Formarea zgomotului speckle pe o suprafață rugoasă cu un LASER și vizualizarea zgomotului [1], [20]

Zgomotul speckle este adesea un factor limitativ în sistemele de imagistică, cum ar fi radarul, radarul cu apertură sintetică (SAR), ultrasunetele medicale și tomografia optică de coerență. De exemplu, în oceanografia SAR, speckle este cauzat de interferența semnalelor provenite de la dispersorii elementari, valurile gravitaționale-capilare, și se manifestă ca o imagine de fond (zgomot de fond), sub imaginea valurilor mării.[1]

Din punct de vedere practic, zgomotul speckle este un zgomot multiplicativ, mai exact acesta este multiplicat cu imaginea originală de intensități, în loc să fie adunat sau scăzut din aceasta. Prin urmare, zgomotul speckle diferă de zgomotul aditiv (ex. zgomotul aditiv alb Gaussian) care este pur și simplu adăugat peste valoarea fiecărui pixel din imagine.

Modelul (en. pattern) speckle este rezultatul interferențelor mai multor unde ce au aceeași frecvență, dar faze diferite, și care sunt împrăștiate din diferite părți ale unui obiect. Când aceste unde interferează unele cu celelalte, fazele adunate ale acestora sunt aleatoare, producând un model complex cu o statistică a intensității luminoase bine definite.

Intensitatea I a modelului speckle, când numărul de unde ce interferează este mare, urmează o funcție a densități de probabilitate exponențială dată de:

$$P(I) = f(I \mid \lambda) = \lambda e^{-\lambda \cdot I}$$

unde λ este parametrul ce determină rata de creștere a exponențialei.

Zgomotul speckle este un fenomen complex care poate fi atât un obstacol, cât și un instrument util în diverse aplicații tehnice și științifice. Acesta poate reprezenta și unele informații utile, în special atunci când este legat de speckle-ul laser și de fenomenul speckle dinamic, unde schimbările modelului spațial speckle în timp pot fi utilizate ca o măsură a activității suprafeței. În anumite tehnici de imagistică medicală și în seismologie, modelul speckle poate furniza informații despre proprietățile obiectului care este fotografiat. De exemplu, acest lucru este util pentru măsurarea câmpurilor de deplasare prin corelația imaginilor digitale.

1.2. Zgomotul Speckle în imaginile SAR

Imaginile SAR (en. Synthetic Aperture Radar - ro. Radar cu Apertură Sintetică) captate de Sentinel-1 sunt adesea afectate de zgomotul speckle.[2] Acest zgomot este cauzat de interferența aleatorie a undelor reflectate de la numeroși dispersori elementari.[3] Speckle-ul în imaginile SAR complică interpretarea imaginilor prin reducerea eficacității metodelor de segmentare și clasificare a imaginilor.[3] Prin urmare, zgomotul speckle este un efect nedorit în cazul monitorizării vărsărilor de petrol în oceane din imaginile satelitare SAR, ce reduce calitatea imaginii și poate masca trăsăturile unui potențial accident petrolier.

Pentru putea a lucra corespunzător cu datele SAR, acestea sunt mai întâi calibrate.[2] Calibrarea radiometrică convertește intensitatea retrodifuzării, așa cum este recepționată de senzor, la secțiunea transversală radar normalizată (Sigma 0), o măsură calibrată care ține cont de unghiul de incidență global al imaginii și alte caracteristici specifice senzorului.[2] Acest lucru este deja realizat pe setul de imagini descărcat în vederea detecției nesupervizate a vărsărilor de petrol în largul mărilor și oceanelor.

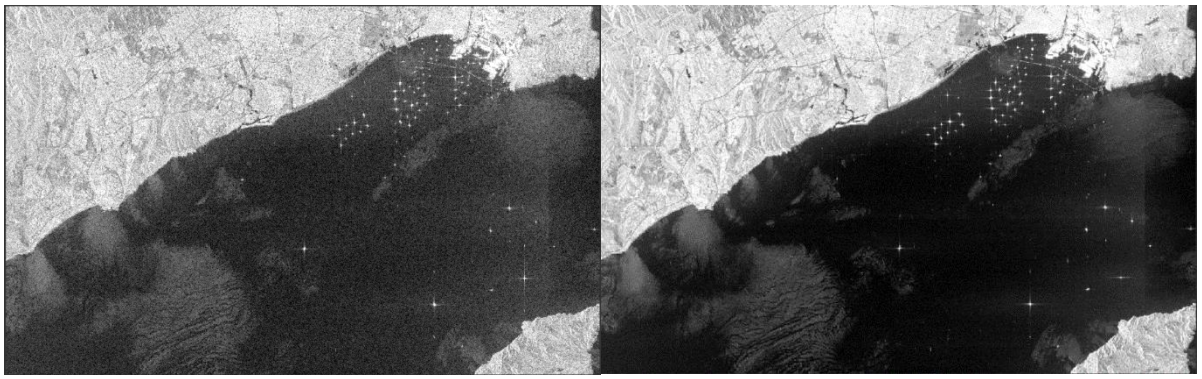
Rămâne ca, în ultimul pas, efectul speckle să fie redus pe cât de mult posibil pentru a evidenția și mai bine trăsăturile suprafeței oceanice. Acest lucru determină scopul lucrării de față, și anume implementarea unei metode de filtrare a zgomotului speckle din imaginile SAR de înaltă rezoluție, într-o manieră cât mai eficientă computațional, utilizând metode de accelerare paralelă a calculelor impusă de dispozitivele GPU și platforma CUDA.

Filtrele speckle pot fi aplicate datelor pentru a reduce cantitatea de zgomot speckle, deși acest lucru se face în detrimentul caracteristicilor estompate sau a rezoluției reduse.[2] Există diverse tehnici de procesare a imaginilor pentru a elimina zgomotul de tip speckle, cum ar fi filtrul Lee sau Lee-Sigma. [4]

Cu toate acestea, trebuie reținut că, în ciuda faptului că zgomotul speckle poate complica interpretarea și procesarea imaginilor SAR, acesta poate oferi, de asemenea, informații valoroase în anumite scenarii. De exemplu, modelele de speckle dinamic, unde schimbările modelului spațial speckle în timp pot fi utilizate ca o măsură a activității suprafeței.[2]

Însă în cazul sarcinii monitorizării suprafeței oceanului în vederea detectării accidentelor maritime ce au ca rezultat vărsări de petrol, zgomotul speckle nu prezintă utilitate, fiind necesară detectarea schimbării proprietăților fizico-chimice ale suprafeței apei și nu a dinamicității geometrice a acesteia.

În contextul detectării scurgerilor de petrol, prezența zgomotului speckle poate face dificilă detectarea cu precizie a marginilor scurgerii de petrol, iar, în unele cazuri, poate duce la detectarea falsă (en. false alarm) a unei scurgeri de petrol. Prin urmare, este adesea benefică aplicarea unei tehnici de îndepărtare a zgomotului speckle pentru a îmbunătăți calitatea imaginilor SAR.



Figură 1.2. Imaginea subiect cu (stânga) și fără (dreapta) zgomot speckle

De exemplu, într-un cadru bazat pe învățare profundă (en. Deep Learning) pentru identificarea scurgerilor de petrol, zgomotul speckle din imagine poate fi eliminat folosind un filtru frost sau lee-sigma. Acest lucru ajută la evitarea detectării false a unei scurgeri de petrol. Cu toate acestea, este important de remarcat că există un compromis între reducerea speckleului și păstrarea detaliilor fine în imaginea SAR fără speckle. Provocarea constă în reducerea zgomotului speckle fără a pierde detaliile fine ale imaginii.

1.3. Arhitecturile CPU și GPU

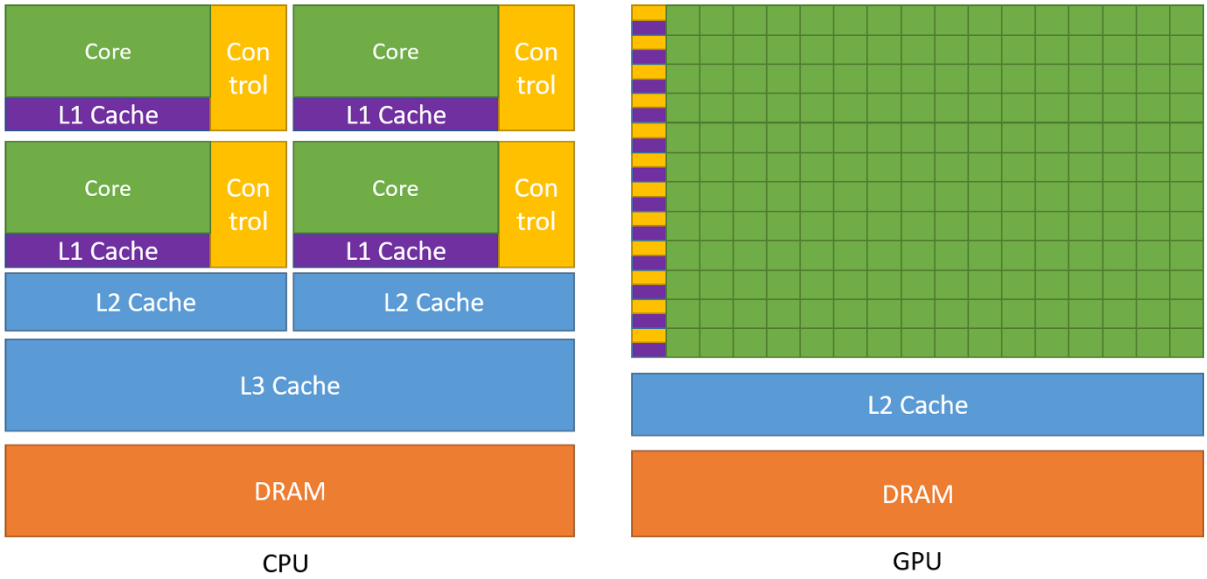
În tehnica de prelucrare computațională a imaginilor, există două platforme posibile ce pot fi utilizate în vederea executării calculelor linear algebrice necesare, unitatea centrală de prelucrare (en. Central Processing Unit – CPU), scurt procesor, sau unitatea grafică de prelucrare (en. Graphics Processing Unit – GPU), sau scurt placa grafică.

Arhitectura CPU este bine cunoscută pentru capacitatea sa de a executa o gamă largă de instrucțiuni și pentru performanța sa în sarcini secvențiale. CPU-urile sunt optimizate pentru a executa câteva fire de execuție cât mai rapid posibil, ceea ce le face ideale pentru sarcini care necesită multă putere de procesare, dar nu pot fi paralelizate eficient. În general, un procesor în zilele noastre deține de la 2-4 nuclee (en. cores) până la 6, 8, 12 sau chiar 16 nuclee, dacă discutăm doar despre procesoarele destinate consumatorilor de rând. Fiecare procesor deține adesea 2 fire de execuție (en. Threads) pentru fiecare nucleu, rezultând într-un număr destul de mare de astfel de fire, fiecare executând o anumită operație a unui program.

Când vine vorba de sarcinile pe care suntem nevoiți să le îndeplinim în viața de zi cu zi folosind sistemele noastre de calcul, sarcini ce necesită operații mai mult sau mai puțin complexe, aceste procesoare prin arhitectura și tipul de instrucțiuni cu care sunt dotate fac ca performanțele să fie extrem de satisfăcătoare. Însă, când volumul de calcul devine extrem de ridicat în cadrul unei simple sarcini precum modificarea valorilor unei matrici după o regulă simplă algebrică, procesorul începe să întâmpine încetiniri semnificative din pricina numărului

redus de fire de execuție. Acesta este unul dintre marile motive pentru care jocurile video, simularea graficii 3D pe ecranul calculatorului, este realizată cu ajutorul unui alt dispozitiv specializat numit placa grafică sau GPU.

Arhitectura GPU (en. Graphics Processing Unit) este proiectată pentru a fi extrem de eficientă în executarea operațiilor algebrice simple, ce pot fi paralelizate. Acestea au mult mai multe nuclee de procesare decât CPU-urile, ceea ce le permite să execute simultan un număr mare de operații. Acest lucru le face ideale pentru sarcini care pot fi divizate în multe sub-sarcini mici care pot fi executate în paralel, cum ar fi procesarea imaginilor sau calculele algebrice liniare (calcule matriceale).



Figură 1.3. Arhitecturile CPU - 4 nuclee și GPU[5]

GPU-urile fiind deosebit de eficiente în calculele algebrice liniare datorită capacității lor de a executa operații în paralel, le face un foarte bun pretendent pentru procesarea imaginilor satelitare de înaltă rezoluție, precum cele SAR. De exemplu, atunci când se efectuează o operație de înmulțire a matricelor (imaginea digitală prelucrată ca matrice de pixeli), fiecare element din matricea rezultată poate fi calculat independent de celelalte. Acest lucru înseamnă că operația poate fi divizată într-un număr mare de sub-sarcini care pot fi executate simultan de nucleele GPU. Acest lucru poate duce la o accelerare semnificativă a calculelor algebrice liniare comparativ cu execuția acestora pe un CPU.

Ceea ce trebuie extras din acest paragraf este că o unitatea de procesare grafică oferă o viteză de procesare a instrucțiunilor și o lărgime de bandă a memoriei mult mai mare decât cea a procesorului. Această diferență de capacități între GPU și CPU există deoarece acestea au fost proiectate cu scopuri diferite. În timp ce CPU este proiectat să excleze în executarea cât mai rapidă a unei secvențe de operații (thread) și poate executa câteva zeci de astfel de thread-uri în paralel, GPU este proiectat să excleze în executarea a mii de thread-uri în paralel.[5]

1.4. Arhitectura CUDA

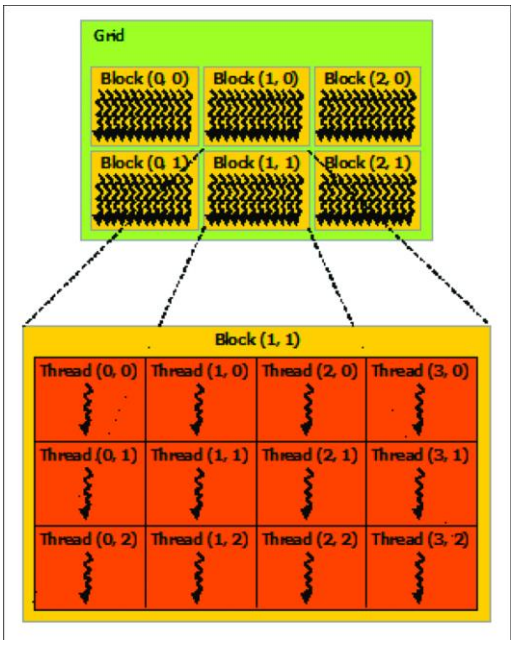
Arhitectura CUDA (en. Compute Unified Device Architecture) este o platformă de calcul paralel și un model de interfață de programare a aplicațiilor (en. Application Programming Interface - API) dezvoltat de NVIDIA.[6] Este o extensie a programării în limbajele C/C++ ce permite paralelizarea algoritmilor și rularea acestora pe arhitecturile paralele GPU ale companiei NVIDIA.[6]

Mediul de dezvoltare software CUDA oferă toate instrumentele, exemplele și documentația necesare pentru a dezvolta aplicații care să beneficieze de puterea de calcul a arhitecturii CUDA.[7] Acesta include biblioteci avansate care includ cuBLAS (en. Basic Linear Algebra), cuRAND pentru generarea de modele aleatoare, cuSOLVER, cuSPARSE pentru operațiile cu matrici sparse, cuFFT, cuDNN pentru rețele neurale adânci și alte funcții optimizate pentru arhitectura CUDA.[8]

CUDA oferă o serie de avantaje față de API-urile tradiționale de prelucrare a datelor cu ajutorul procesoarelor video. De exemplu, permite citiri nesecvențiale din locații de memorie arbitrare, oferă memorie partajată de mare viteză care poate fi împărțită între firele de execuție și suportă operațiile complete cu numere întregi și pe bit.

Ca limitări, CUDA suportă doar un subset de operații al limbajului de programare C și o serie de extensii simple. Lipsesc pointerii la funcții și recursivitatea. Iar în cazul dispozitivelor dezvoltate pe arhitectura Fermi, în codul compilat pot fi folosite clase C++ atâta timp cât nicio funcție membră a clasei nu este virtuală.

Programele paralelizate și executate folosind API-ul CUDA se numesc Kernel-uri (ro. nucleu, miez). Fiecare Kernel se execută folosind fire de execuție paralele, acestea fiind de ordinul miilor, iar arhitectura implementată se numește SIMD (Single Instruction, Multiple Data). Mai exact, asupra unei masive de date, îi este aplicată o singură funcție algebrică în mod repetat, lucru care se poate asemăna foarte bine cu operația de filtrare (ajustare) a pixelilor unei imagini (matrici de dimensiuni mari).



Figură 1.4. Arhitectura CUDA[9]

Arhitectura CUDA ce este ilustrată în Figură 1.4, are la bază thread-ul (firul de execuție), fiecare dintre acestea corespunzând unui kernel de execuție CUDA. Mai departe, un grup de thread-uri constituie un block iar block-urile sunt grupate la rândul lor într-un grid, fiecare grid corespunzând unui kernel de execuție. Grid-urile și block-urile pot fi unidimensionale, bidimensionale sau tridimensionale, cele din figură fiind bidimensionale. Presupunând exemplul din figură, fiecare block este alcătuit din 12 thread-uri (dispuse 4 x 3) iar fiecare grid este alcătuit din 6 blocuri (dispuse 3 x 2), vom avea în total 72 de fire de execuție.

2. Metoda abordată

În acest paragraf voi descrie metoda abordată pentru eliminarea zgomotului speckle din imaginile de teledetecție Sentinel-1 în vederea utilizării lor ulterioare în cadrul unui proiect menit să antreneze o rețea neurală adâncă nesupervizată pentru detecția poluării cu petrol a oceanelor și a mărilor. Paragraful va descrie matematica din spatele filtrului Lee-Sigma, metoda prin care acesta este aplicat pe imaginile SAR de teledetecție, din punct de vedere computațional, precum și metodele de implementare în limbajul ales și bibliotecile disponibile.

2.1. Filtrul Lee-Sigma

Filtrul Lee-Sigma, alături de multe alte filtre similare, este un instrument eficient pentru reducerea zgomotului speckle, un fenomen comun în imaginile radar cu apertură sintetică.[10] Filtrul Lee-Sigma funcționează prin aplicarea unui filtru spațial fiecărui pixel dintr-o imagine.[10] Acest filtru prelucrează datele pe baza statisticilor locale calculate într-o fereastră pătrată, de dimensiune dată.[10] Valoarea pixelului central este înlocuită cu o valoare calculată folosind statistica pixelilor vecini.[10]

Prin urmare, întregul proces constă în aplicarea unui filtru spațial fiecărui pixel dintr-o imagine. Acest filtru procesează datele pe baza statisticilor locale calculate într-o fereastră pătrată, iar valoarea pixelului central este înlocuită cu o valoare calculată folosind pixelii vecini.

Mai exact, filtrul Lee-Sigma utilizează o abordare bazată pe statistici pentru a estima valoarea unui pixel în funcție de valorile pixelilor din vecinătatea sa. Acesta calculează media și deviația standard a pixelilor din fereastra locală și apoi utilizează aceste valori pentru a determina dacă un pixel este zgomot sau face parte dintr-o caracteristică a imaginii. Dacă un pixel este identificat ca zgomot, acesta este înlocuit cu media pixelilor din fereastra locală. Dacă un pixel este identificat ca făcând parte dintr-o caracteristică a imaginii, acesta este lăsat neschimbat.

Filtrul Lee-Sigma este un filtru de tip MMSE (Minimum Mean Square Error) care se aplică pe o fereastră locală a imaginii. Pe scurt, pașii matematici urmați în vederea filtrării zgomotului speckle din imaginile SAR sunt următorii:

1. Se calculează media (*local_mean*) și varianța (*local_var*) a pixelilor din fereastra locală extrasă din imagine (*vec*) astfel:

$$\text{local\_mean} = \frac{1}{N}\sum_{i=1}^{N} x_i \qquad \text{local\_var} = \frac{1}{N}\sum_{i=1}^{N} (x_i - \text{local\_mean})^2$$

unde x_i este valoarea pixelului i din fereastră, iar N este numărul total de pixeli din fereastră.

2. Se estimează varianța zgomotului (*noise_var*) ca produs al unei constante universale (*Cu*) și a mediei locale calculate la pasul anterior.

$$\text{noise\_var} = Cu \cdot \text{local\_mean}$$

3. Se calculează coeficientul de variație pătrat (*sigma_squared*), care este determinat de varianța locală împărțită la pătratul mediei locale.

$$\sigma^2 = \frac{\text{local\_var}}{(\text{local\_mean})^2}$$

4. Dacă coeficientul local de variație pătrat (*sigma_squared*) este mai mic sau egal cu *Cmax* (un coeficient maxim pentru varianța zgomotului), se calculează o pondere (*weight)* care este 1 minus raportul dintre varianța zgomotului și varianța locală. Această pondere este apoi limitată între 0 și 1. Altfel, dacă coeficientul local de variație pătrat este mai mare decât *Cmax*, ponderea este setată la 0.

$$\text{weight} = \begin{cases} 1 - \frac{\text{noise\_var}}{\text{local\_var}}, & \text{dacă } \sigma^2 \leq Cmax \\ 0, & \text{altfel} \end{cases}$$

5. În final, se calculează valoarea filtrată ca media locală înmulțită cu ponderea plus valoarea pixelului central înmulțită cu (1 - pondere).

$$\text{filtered\_value} = \text{local\_mean} \cdot \text{weight} + x_{\text{center}} \cdot (1 - \text{weight})$$

unde *x_center* este valoarea pixelului central din fereastra locală (pixelul filtrat).

Ideea din spatele filtrului Lee-Sigma constă în determinarea proprietăților statistice ale fiecărei ferestre, iar în funcție de valoarea medie și deviația standard a acesteia, se va lua o decizie în privința pixelului central ce trebuie fie filtrat sau lăsat neschimbat. În cazul în care pixelul se abate mult de la aceste statistici, el va fi ajustat astfel încât să se încadreze în acestea.

Această abordare permite filtrului Lee-Sigma să reducă zgomotul speckle, păstrând în același timp detaliile fine ale imaginii. Însă pentru a putea păstra în mod corect detaliile, este necesară alegerea corectă a valorilor constantei universale *Cu* și a coeficientului maxim de variație *Cmax.*

Constanta universală (*Cu*) este o constantă folosită pentru a scala modelul de zgomot în filtrul Lee-Sigma. Cu alte cuvinte, *Cu* este folosit pentru a estima varianța zgomotului ca un produs al constantei universale și a mediei locale. Acest lucru permite filtrului să ajusteze nivelul de zgomot în funcție de intensitatea semnalului local. Valoarea lui *Cu* este de obicei aleasă empiric și poate varia în funcție de caracteristicile imaginii și de nivelul de zgomot speckle.

Iar coeficientul maxim de variație a zgomotului (*Cmax*) este un prag ce definește limita pentru varianța zgomotului. Dacă coeficientul de variație pătrat (*sigma_squared*) este mai mic sau egal cu *Cmax*, atunci filtrul consideră că pixelul este afectat de zgomot și aplică o pondere pentru a reduce acest zgomot. Dacă *sigma_squared* este mai mare decât *Cmax*, atunci filtrul consideră că pixelul face parte dintr-o caracteristică a imaginii și nu aplică nicio pondere, lăsând pixelul neschimbat. Prin urmare, Cmax este folosit pentru a diferenția între zgomot și caracteristicile imaginii.

Ambii coeficienți sunt esențiali pentru funcționarea filtrului Lee-Sigma și permit filtrului să se adapteze la diferite niveluri de zgomot și caracteristici ale imaginii. Acest lucru,

face ca filtrul să fie extrem de util în multe aplicații, în special în procesarea imaginilor SAR, unde zgomotul speckle poate fi un impediment.

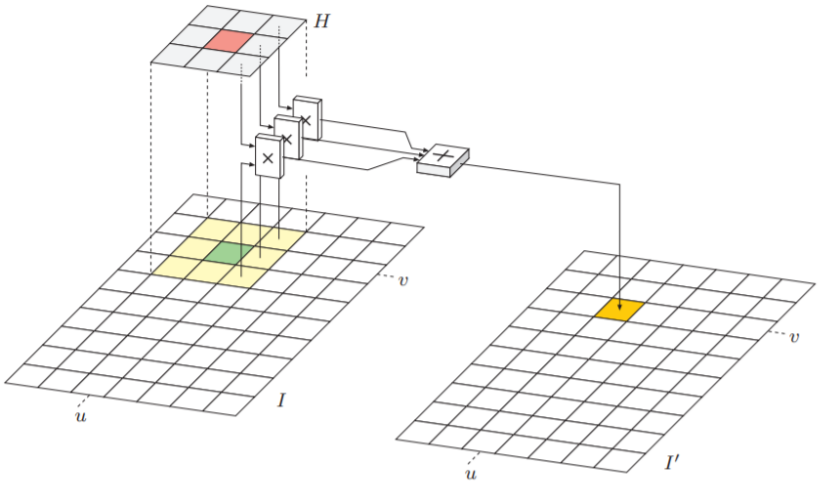
2.2. Implementarea algoritmică a filtrului Lee-Sigma

Știind că sarcina finală a imaginilor SAR este de a fi utilizate în antrenarea unei rețele neurale adânci în vederea identificării anomaliilor caracterizate prin vărsări de petrol, se va aborda o implementare a filtrării zgomotului speckle cu ajutorul limbajul Python, pentru a putea lega acest beneficiu cu implementare rețelei neurale cu PyTorch.

Bibliotecile puse la dispoziție în limbajul Python ce dețin accelerare GPU sunt Numba și CuPy, varianta pentru arhitectura CUDA a bibliotecii NumPy. Astfel, implementările realizate folosind Numba și CuPy vor fi comparate cu implementarea folosind NumPy, ce beneficiază doar de accelerarea CPU (multithreading).

O implementare pură în Python folosind doar structuri Python, precum listele, nu ar avea sens deoarece acestea nu beneficiază de optimizări CPU rulând doar pe un singur fir de execuție. Timpii rulare unei astfel de implementări pot ajunge la minute întregi ținând cont că imaginile satelitare sunt de înaltă rezoluție (zeci - sute de mii de pixeli sau chiar milioane).

Din punct de vedere computațional, implementarea algoritmului este foarte similară cu definiția matematică a filtrului. O fereastră de dimensiune prestabilită este glisată, pe rând, în imaginea SAR, îi sunt calculate proprietățile statistice ale informației și zgomotului ca, mai apoi, să se ia o decizie în privința filtrării pixelului central. Acest algoritm ridică totuși o problemă, și anume modalitatea de tratare a pixelilor aflați la marginea imaginii ce nu pot fi înconjurați perfect cu pixeli vecini într-o fereastră locală pătrată. În cazul acestor pixeli am ales să îi las neschimbați și să îi exclud din procesul de filtrare. Astfel, dimensiunea finală a imaginii rămâne neschimbată, iar informația nu este afectată printr-o bordare a marginilor cu valori de 0. Pixelii marginali nu prezintă oricum un interes semnificativ în viitoarea sarcină, astfel că păstrarea valorilor neschimbate nu va afecta performanța rețelei neurale.



Figură 2.1. Filtrarea unei matrici cu un filtru 3x3[11]

Ținând cont de toate aceste observații, o primă concluzie ce poate fi extrasă despre acest algoritm este că este mai precis și mai eficient cu cât dimensiunea ferestrei filtrului este mai mare. O dimensiune mai mare a acestui filtru se traduce într-un număr mai mare de eșantioane (i.e. pixeli) în compunerea statisticii locale a imaginii și, prin urmare, o generalizare mult mai bună a parametrilor medie și dispersie. Dar, un filtru de dimensiune mai mare va duce la o reducere și mai mare a dimensiunii finale a imaginii filtrate (numărul de pixeli ce compun marginea imaginii și nu pot fi filtrați, crește). Dimensiunea înaltă a imaginilor satelitare și centrarea conținutului acestora pe zona de interes va putea permite sacrificarea unui număr suficient de mare de pixeli marginali.

În paragraful următor, voi detalia modalitatea de implementare a funcției de filtrare precum și limitările întâmpinate în funcție de biblioteca utilizată.

2.3. Implementarea filtrării Lee-Sigma folosind NumPy – CPU

NumPy este pachetul fundamental pentru efectuarea calculelor științifice în Python. Este o bibliotecă Python care oferă un obiect pentru matricea multidimensională, diverse obiecte derivate și o varietate de rutine pentru operații matriceale rapide, inclusiv matematice, logice, manipulare a formelor, sortare, selectare, I/O , transformate Fourier discrete, algebră liniară de bază, operații statistice de bază, simularea aleatoriei și multe altele. Această bibliotecă este imperios necesară în vederea implementării operației de filtrare a imaginilor SAR, atât pentru operațiile algebrice liniare pe care le pune la dispoziție, dar mai ales pentru accelerarea CPU de care beneficiază.

Atât implementarea folosind NumPy, cât și toate celelalte implementări ce beneficiază de biblioteci cu accelerare GPU au o etapă inițială de încărcare a imaginii satelitare. Această imagine este încărcată în prealabil cu biblioteca esa_snappy, într-un alt script, și este salvată în formatul „.npy” destinat matricelor NumPy. Acest lucru va facilita încărcarea imaginii SAR, eliminând partea de citirea a metadatelor acesteia și eliminând timpul pierdut cu inițializarea bibliotecii mult mai complexe implementată de ESA.[12]

De asemenea, în toate tipurile de implementare este prestabilită dimensiunea ferestrei locale de filtrare, precum și capătul imaginii, omițând pixelii marginali ce nu pot fi filtrați. Capătul de filtrare al imaginii este egal cu dimensiunea filtrului împărțit la 2 și rotunjit prin lipsă. Mai precis, pentru un filtru de dimensiune (3, 3), capătul de filtrare va fi 3 // 2 = 1, deci imaginea va fi parcursă de capăt (1) până la h-capăt, respectiv de la capăt până la w-capăt.

Implementarea NumPy se folosește de capacitatea structurilor de date np.arrays (matrici numpy – np.) de a beneficia de accelerare în timpul operațiilor algebrice liniare de bază. Astfel, în funcția *apply_lee_sigma_filter* sunt trimiși ca parametrii imaginea originală, dimensiunea filtrului, constanta universală și coeficientul maxim de variație al zgomotului.

Funcția *apply_lee_sigma_filter* inițializează un nou np.array de dimensiunea imaginii cu valori 0 în care vor fi stocate valorile trecute prin funcția de filtrare a pixelilor. În următoarea etapă, folosind două bucle for, una pentru lățimea și alta pentru înălțimea imaginii, sunt parcurși toți pixelii imaginii conform limitelor descrise anterior. Sunt extrase ferestre de dimensiunea

[i - capat:i + capat + 1, j - capat:j + capat + 1] pentru fiecare pixel centrat ce urmează a fi filtrat și sunt trimise, pe rând, ca parametru în funcție finală de filtrare *lee_sigma_filter*.

Funcția *lee_sigma_filter* mai primește ca parametru dimensiunea ferestrei, constanta universal și coeficientul maxim de variație al zgomotului. În interiorul funcției sunt implementate toate operațiile matematice descrise în paragraful 2.1., exact în ordinea în care sunt prezentate. Toate operațiile sunt realizate folosind funcții disponibile în biblioteca NumPy.

2.4. Implementarea filtrării Lee-Sigma folosind Numba – GPU

Numba este un compilator JIT (Just-In-Time) open-source care traduce un subset din codul Python și NumPy în cod rapid de mașină, utilizând biblioteca de compilare LLVM standard din industrie.[13], [14], [15], [16] Numba poate compila un subset mare din operațiile și tipurile de date Python concentrate pe maipularea datelor numerice, mai ales a funcțiilor linear algebrice din NumPy.[14], [16] Rolul principal al bibliotecii Numba este de a genera cod specializat pentru diferitele tipuri de date și funcții pentru a optimiza performanța la rulare a programului implementat.[13]

Numba oferă, de asemenea, o gamă diversă de opțiuni pentru paralelizarea codului pentru CPU și GPU, adesea doar cu modificări minore ale codului. Acceptă programarea CUDA GPU prin compilarea directă a unui subset restrâns de cod Python în nucleele CUDA și în funcțiile dispozitivului, urmând modelul de execuție CUDA. Kernel-urile scrise în Numba au acces direct la matrice NumPy, mai exact la zonele lor de memorie prin încărcarea datelor pe dispozitivul CUDA folosind o funcție specifică Numba, „*cuda.to_device*()”. Pot fi create, de asemenea, matrici NumPy direct pe dispozitivul CUDA folosind comanda „*cuda.device_array_like*()”. Iar, în final, rezultatul Kernel-ului poate fi copiat înapoi pe dispozitivul CPU folosind metoda „.copy_to_host()”.

O dată cu scrierea nucleelor CUDA cu Numba, soluția implementată va fi modelată prin definirea unei ierarhii de fire de execuție formată din grile, blocuri de fire de execuție și fire de execuție. Suportul CUDA al Numba expune facilități pentru declararea și gestionarea acestei ierarhii de fire de execuție.

Cunoscând aceste capacități ale bibliotecii Numba, tot ce a rămas de făcut a fost modificarea funcției anterioare ce implementa filtrarea Lee-Sigma, „*lee_sigma_filter*”, într-o funcție Kernel. Din interiorul funcției anterioare „*apply_lee_sigma_filter*”, sunt generate Kernel-uri, în funcție de numărul de fire de execuție per bloc și numărul de blocuri per grilă (en. grid). Numărul de thread-uri per bloc va fi prestabilit, astfel încât să nu depășesc limita maximă de 1024 thread-uri, iar numărul de blocuri per grilă va fi determinat astfel încât să fie acoperită întreaga imagine cu filtre. Numărul de thread-uri per bloc va fi declarat bidimensional, în concordanță cu structura matriceală a imaginilor SAR cu un singur canal.

Funcția de filtrare „*lee_sigma_filter*” devine o funcție Kernel prin atribuirea decoratorului *@cuda.jit* menit să indice compilarea codului ca funcție pentru GPU. Noua funcție Kernel implementată primește o nouă etapă de verificare a indecșilor globali ai thread-ului, ținând cont de faptul că numărul de thread-uri este fix per bloc, iar numărul de blocuri este generat astfel încât să acopere întreaga imagine cu thread-urile conținute. Astfel, vor exista

și thread-uri ce depășesc dimensiunile imaginii. Prin urmare, cu ajutorul identificatorilor globali ai fiecărui thread se va verifica dacă acesta se află între dimensiunile maxime pe cele două axe ale imaginii, iar acești indecși vor servi mai departe ca indecși ai pixelului central ce urmează a fi prelucrat.

Din acest punct, pe baza indecșilor thread-ului este determinată dimensiunea ferestrei de filtrare, astfel încât să fie maxim de dimensiunea prestabilită sau suficient de mare încât să nu depășească limitele imaginii, în cazul pixelilor marginali. În acest scenariu, funcția va filtra și pixelii marginali, însă în ferestre de dimensiuni mai mici decât cele stabilite inițial. A fost aleasă această abordare pentru a se ținti către o filtrare și mai eficientă a zgomotului speckle, pe lângă posibila accelerare a întregii operații. În implementarea inițială pixelii marginali au fost omiși din operația de filtrare pentru a nu impune și mai multe operații suplimentare de redimensionare a ferestrei locale în timpul filtrării.

Restul operațiilor de determinare a proprietăților statistice locale ale fiecărei ferestre și a valorii ponderii de filtrare rămân neschimbate față de implementarea anterioară.

2.5. Implementarea filtrării Lee-Sigma folosind CuPy – GPU

CuPy este o bibliotecă destinată operațiilor matriceale, open-source, ce beneficiază de calcul accelerat de GPU în Python.[17], [18], [19] Este conceput pentru a fi compatibil cu NumPy și SciPy și, în cele mai multe cazuri, poate fi folosit ca înlocuitor direct al acestor biblioteci.[17], [18], [19] Acest lucru înseamnă că orice cod scris deja cu NumPy/SciPy poate fi rulat pe platformele NVIDIA CUDA sau AMD ROCm înlocuind pur și simplu numpy și scipy cu cupy și cupyx.scipy în codul Python deja implementat.[17]

CuPy utilizează biblioteci CUDA Toolkit, inclusiv cuBLAS, cuRAND, cuSOLVER, cuSPARSE, cuFFT, cuDNN și NCCL pentru a folosi pe deplin arhitectura GPU.[17], [18] Acest lucru îi permite lui CuPy să atingă performanțe ridicate în calculele la scară largă, depășind adesea calculele echivalente bazate pe CPU.[17] Prin urmare, CuPy reprezintă un foarte bun pretendent pentru accelerarea calculelor necesare filtrării imaginii SAR de înaltă rezoluție profitând de puterea de calcul a arhitecturii CUDA.

Datorită modificărilor minime necesare ce trebuie efectuate asupra codului NumPy deja prezentat în paragraful 2.3., adoptarea bibliotecii CuPy este mult mai facilă decât a bibliotecii Numba. Însă această facilitate datorită similarității acestor biblioteci vine cu un cost și anume cel al flexibilității implementării algoritmului paralel, nemaiputând încărca întreg codul cu toate operațiile sale algebrice liniare pe GPU sub forma unui Kernel. De asemenea, nu mai poate fi controlat gradul de paralelism și numărul de fire de execuție lansate pentru a rezolva sarcina.

Cu toate acestea, CuPy acceptă și Kernel-uri definite de utilizator, permițând scrierea de cod CUDA personalizat în Python și executarea acestuia.[17] Însă această operație este deja suportată într-un mod mult mai eficient de către Numba. Rularea unui Kernel personalizat poate fi utilă atunci când este obligatorie o optimizarea a anumitor părți ale codului sau când există deja o implementare a funcției dorite într-un astfel de Kernel.

Față de implementarea NumPy a filtrului Lee-Sigma, în implementarea CuPy este, din nou, necesar transferul matricii NumPy de pe CPU pe GPU cu funcția „*cp.asarray(img)*”. Pe lângă această operație, este generat un nucleu de mediere a ferestrei curente și este bordată fereastra curentă cu valorile oglindite ale imaginii în lungul axei curente, astfel încât marginile ferestrei să poată fi și ele incluse în procesul de mediere al ferestrei locale. Media și media pătratică sunt determinate printr-o operație de convoluție între fereastra bordată și filtrul de mediere, iar varianța este determinată ca diferența acestor două momente statistice. În final, după calculul variației zgomotului și a valorii coeficientului de variație la pătrat (en. sigma squared), este aplicată regula de determinare a valorii ponderilor pentru fiecare pixel din imagine. Aceste ponderi sunt apoi aplicate peste imaginea originală conform formulei 5, sub forma unor operații matriceale, element cu element.

3. Desfășurarea experimentelor și rezultate experimentale

În următoarele două paragrafe voi prezenta desfășurarea a trei experimente în vederea determinării celei mai eficiente biblioteci și a celor mai eficienți parametrii pentru generarea firelor de execuție în implementările CUDA ale operației de filtrare.

Primul experiment va consta în creșterea treptată a dimensiunii imaginii SAR și observarea efectelor acestor creșteri asupra timpului de rulare, în funcție de arhitectura utilizată. Acest lucru va permite determinarea accelerării diferitelor implementări, în funcție de dimensiunea datelor prelucrate. Un experiment adițional va fi derulat pentru a determinat influența dimensiunii ferestrei de filtrare asupra timpului de rulare, în funcție de biblioteca utilizată.

Ultimul experiment va consta în alegerea unei dimensiuni fixe a imaginii SAR și schimbarea numărului de fire de execuție din blocuri și a numărului de blocuri din grile pentru a determina parametrii ideali pentru un timp de rulare și număr de resurse utilizate minim.

Experimentele se vor desfășura pe laptopul personal, folosind un procesor Intel Core I5 10500H ce deține 6 nuclee, 12 fire de execuție, împreună cu o placă grafică NVIDIA RTX 3050Ti ce deține 4GB memorie VRAM GDDR6 și 2560 nuclee CUDA.

3.1. Măsurarea performanțelor bibliotecilor în funcție de dimensiunea datelor

Cum am prezentat și în introducerea capitolului, primul experiment va urmări determinarea celei mai rapide biblioteci din punct de vedere computațional și se va măsura accelerarea dintre aceste implementări. Imaginea SAR selectată și prezentată în Figură 1.2. este de o rezoluție 5297x8595 pixeli înălțime și lățime. Experimentele se vor desfășura pentru subseturi din această imagine de 600x800 pixeli, 800x800 pixeli, 1000x1000 pixeli, 1000x1500 pixeli și, în final 1680 x 2000 pixeli. Fereastra de filtrare ce va fi utilizată va fi de dimensiune 3x3.

Pentru rularea codului pe arhitectura Numba, au fost generate (16, 16) fire de execuție și, în funcție de rezoluția imaginii, un număr de blocuri în grilă suficient de mare încât să acopere întreaga imagine.

Tabel 1. Performanța fiecărei implementări, în funcție de dimensiunea imaginii

Dimensiunea imaginii	Timp rulare NumPy	Timp rulare Numba	Accelerare Numba	Timp rulare CuPy	Accelerare CuPy
600 x 800	16.72 s	1.26 s	x13.26	1.07 s	x15.62
800 x 800	23.76 s	0.61 s	x39.07	0.41 s	x58.23
1000 x 1000	35.16 s	0.61 s	x57.63	0.41 s	x85.75
1000 x 1500	53.13 s	1.25 s	x42.50	0.89 s	x59.69
1680 x 2000	128.21 s	1.16 s	x110.52	1.15 s	x111.48

În urma rezultatelor vizuale obținute, s-a constatat că filtrarea imaginilor de rezoluție mică precum 600x800 pixeli sau 800x800 pixeli nu a fost fezabilă în cazul imaginii afectate de zgomotul speckle. Din pricina numărului redus de pixeli și a ferestrei locale de dimensiune mică, statistica nu a fost suficient de generală pentru a putea realiza o filtrarea efectivă a zgomotului. Dimensiunea redusă a ferestrei de filtrare a fost necesară pentru a putea păstra calitatea detaliilor din imagine. Chiar și folosind o fereastră 3x3, cea mai mică dimensiune posibilă, în cele două imagini menționate își face simțită prezența un efect de mediere (blur) al detaliilor. Prin urmare, o filtrare a unei ferestre de dimensiuni mai mici de 1000x1000 pixeli nu este suficient de satisfăcătoare, chiar dacă zonele de interes ce vor fi utilizate în antrenarea rețelei neurale adânci sunt mult mai mici.

Performanța bibliotecii NumPy pe CPU este, din start, mult mai slabă decât Numba sau CuPy indiferent de dimensiunea imaginii alese și prezentate în tabel. Pentru dimensiunile 600x800, 800x800 și 1000x1000 timpul de executare este decent pentru CPU, atingând maxim 35 de secunde. Însă orice dimensiune mai mare de 1000x1000 pixeli are un timp de rulare ce depășește un minut.

Accelerarea bibliotecii Numba este de până la 30, 50 sau chiar 110 de ori mai mare decât timpul de execuție al bibliotecii NumPy, durând, în medie, aproximativ o secundă filtrarea întregii imagini.

Biblioteca CuPy, mult mai bine optimizată decât Numba, obține accelerări puțin mai mari decât aceasta, însă timpii de execuție nu sunt cu mult mai mici. Orice este sub o secundă timp de execuție este insesizabil comparativ cu timpul necesar implementării NumPy. Implementarea CuPy este mult mai simplistă și mai rigidă decât cea Numba, folosind doar funcții predefinite din bibliotecă. Acest lucru este atât un lucru bun, prin faptul că funcțiile CuPy sunt mult mai bine optimizate pentru platformele Cuda, dar și un dezavantaj din pricina modificărilor semnificative ce trebuie aduse asupra întregii logici a codului pentru a putea implementa o versiune îmbunătățită a filtrului Lee-Sigma.

Pentru o imagine de dimensiune mare, cea de 1000x1000 pixeli, se va varia dimensiunea ferestrei de filtrare pentru a determina efectul acesteia asupra vitezei de procesare.

Tabel 2. Performanța fiecărei implementări, în funcție de dimensiunea ferestrei

Dimensiunea ferestrei de filtrare	Timp rulare NumPy	Timp rulare Numba	Accelerare Numba	Timp rulare CuPy	Accelerare CuPy
3 x 3	35.16 s	0.61 s	x57.63	0.41 s	x85.75
7 x 7	35.21 s	0.49 s	x71.85	0.68 s	x51.77
9 x 9	44.45 s	0.61 s	x72.86	0.41 s	x108.4
13 x 13	35.63 s	1.08 s	x31.13	0.89 s	x37.78

Rezultatele arată că utilizarea unei dimensiuni mai mari pentru fereastra de filtrare nu afectează în mod sesizabil performanța funcției de filtrare, indiferent de arhitectura utilizată. Variațiile de accelerare apar, cel mai probabil, din pricina proceselor existente în fundal ce afectează timpul de execuție al operațiilor CPU (calcule, transfer al datelor pe/de pe GPU etc.).

3.2. Măsurarea performanțelor în funcție de numărul de fire de execuție

În acest paragraf, voi testa efectul numărului de fire de execuție lansate într-un bloc, ținând cont de structura sa bidimensională. Trebuie reținut însă că numărul de blocuri din grid este egal generat astfel încât întreaga imagine să fie acoperită și fiecare pixel să fie procesat de către un thread. În cazul în care numărul de thread-uri nu poate acoperi întreaga imagine, atunci anumiți pixeli vor fi procesați pe rând, după finalizarea celorlalte procese deja în desfășurare.

Dimensiunea imaginii pe care se vor desfășura experimentele este 1000x1000, iar dimensiunea filtrului va fi de 3x3. Ținând cont de timpul de execuție anterior pentru imaginea 1000x1000 pe CPU, se va calcula și accelerarea produsă de către CUDA.

Tabel 3. Performanțele obținute în funcție de numărul de fire de execuție lansate

Număr fire de execuție	Timp Numba	Accelerare Numba
(1, 1)	1.41 s	x24.93
(1, 2) / (2, 1)	1.67 s	x21.05
(3, 3)	1.78 s	x19.75
(4, 4)	1.28 s	x27.46
(8, 8)	1.27 s	x27.68
(10, 10)	0.61 s	x57.63
(16, 8) / (8, 16)	1.19 s	x29.54
(16, 16)	0.62 s	x56.70
(32, 16) / (16, 32)	1.09 s	x32.25
(25, 25)	0.61 s	x57.63
(32, 32)	0.61 s	x57.63

Experimentele au demonstrat teoria, cu cât sunt mai puține fire de execuție într-un bloc, fapt ce duce la generarea mai multor blocuri în grid, crește și timpul de execuție. Numărul suficient de fire de execuție într-un bloc este de (16, 16), orice valoare mai mare de 256 de thread-uri în bloc neaducând îmbunătățiri considerabile în timpul de execuție.

Fiecare bloc nu poate avea mai mult de 512 fire în total pentru capacitatea de calcul (en. compute capability) < 2.0 sau 1024 fire pentru capacitatea de calcul >= 2.03. Dimensiunile maxime ale fiecărui bloc sunt limitate la [512,512,64] pentru Compute 1.x și [1024,1024,64] pentru Compute 2.x sau ulterior.

Un aspect foarte important de remarcat este acela că orice dimensiune disproporționată de număr de thread-uri în structura 2-D a blocului duce la îmbunătățiri prea mici sau chiar înrăutățiri ale performanței. Acest fenomen are loc deoarece aspectul imaginii SAR procesate este unul pătrat, cu număr egal de pixeli atât pe lățime cât și pe înălțime. Alegerea unui număr inegal de fire de execuție nu va duce întotdeauna la îmbunătățirea performanțelor, în unele situații acestea înrăutățindu-se deoarece vor fi generate și fire de execuție ce nu vor procesa un pixel al imaginii (i.e. fire de execuție suplimentare ce nu erau necesare, dar respectă

dimensiunea deja prestabilită). Firele de execuție suplimentar generate vor consuma timp pentru generarea lor, însă fără vreo utilitate procesarea filtrării imaginii.

Exact cum am menționat anterior, cel mai eficient număr de fire dintr-un bloc în arhitectura CUDA poate depinde de mulți factori, inclusiv algoritmul implementat, arhitectura GPU și dimensiunea și forma datelor.

Cu toate acestea, este recomandată alegerea numărului de fire de execuție per bloc astfel încât să fie un multiplu al mărimii warp-ului, care este 32 pe tot hardware-ul NVIDIA curent. Acest lucru se datorează faptului că nucleele emit instrucțiuni în warps (grupuri de 32 de fire). Dacă este aleasă o dimensiune a blocului care nu este un multiplu de 32, GPU-ul va emite în continuare comenzi multiplu de 32 de fire de execuție, iar orice fire de execuție suplimentar va fi inactiv, irosind resursele de calcul. Acest lucru a putut fi observat în Tabel 3, unde pentru dimensiuni ale blocurilor de thread-uri ce nu erau multiplu de 32 viteza de calcul era mult mai scăzută. Cel mai bun exemplu este alegerea dimensiunii (3, 3) unde sunt generate 9 thread-uri per bloc, număr ce nu este multiplu de 32 și unde dimensiunile imaginii nu sunt divizibile cu 3, deși blocul are aspect pătrat.

De asemenea, în Tabel 3 se poate observa că alegerea unei dimensiuni multiplu de dimensiunea datei prelucrate nu este întotdeauna alegerea optimă. Utilizând dimensiunile (10, 10) și (25, 25) am întâmpinat instabilități la rulare, rezultând în anumite momente timpi de execuție foarte buni, iar în alte momente timpii de execuție se dublau față de scenariile rapide.

În ultimul experiment ce îl voi desfășura mă voi concentra pe maximizarea gradului de ocupare în funcție de dimensiunea aleasă a datelor. Se vor alege diferite aspecte și dimensiuni ale imaginii SAR și se va urmări dimensionarea blocurilor/grilelor pentru a se potrivi cu dimensiunea datelor și, simultan, să se maximizeze gradul de ocupare, adică numărul fire active simultan. Factorii majori care influențează ocuparea sunt utilizarea memoriei partajate, utilizarea registrului și dimensiunea blocului de fire.

Tabel 4. Determinarea numărului optim de fire de execuție - (5280, 8576) – Filtru 7x7

Fire per Bloc / Blocuri per Grilă	Timp Numba
(16, 16) / (330, 536)	0.89 s
(32, 8) / (165, 1072)	1.34 s
(8, 32) / (660, 268)	1.34 s
(32, 16) / (165, 536)	1.35 s
(16, 32) / (330, 268)	0.76 s
(32, 32) / (165, 268)	0.78 s

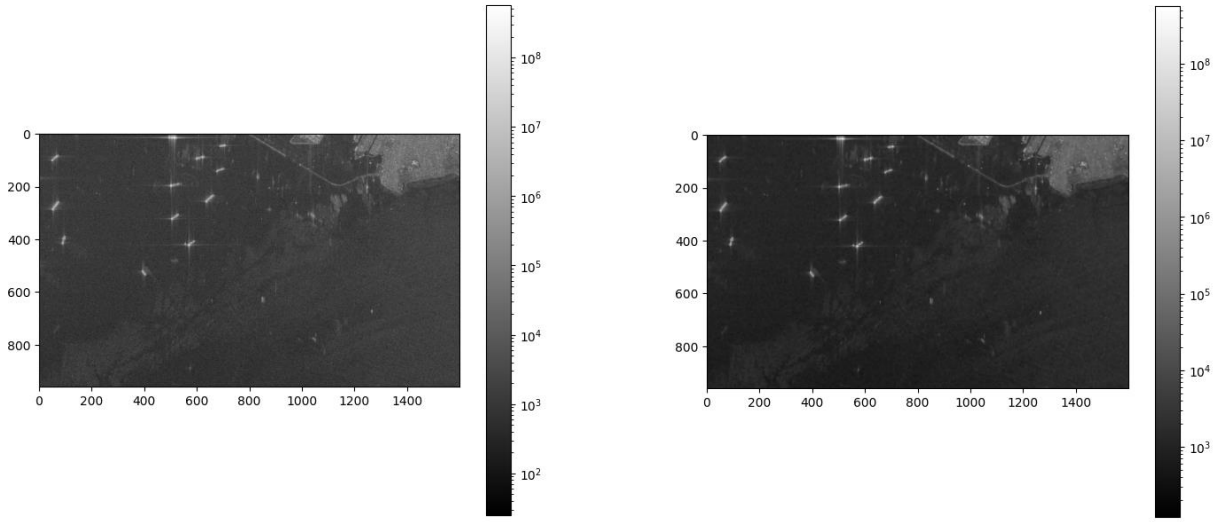
Rezultatele primului tabel afișează situația în care imaginea are dimensiunea maximă, dar dimensiunea acesteia este redusă astfel încât numărul de pixeli pe înălțime și lățime să fie divizibil cu 32 (numărul ideal de thread-uri generate într-un bloc). În acest scenariu timpul de execuție este mult mai bun cu cât numărul de thread-uri este mai mare. În acest caz, pentru că imaginea depășește cu mult numărul maxim de nuclee CUDA pe care le are placa mea grafică – 2560 – aspectul ales pentru blocul de fire de execuție nu are relevanță semnificativă asupra performanței finale. Fluctuațiile valorilor rezultate sunt date de starea sistemului pe care au fost

rulate testele. Timpul de execuție pentru această dimensiune a imaginii cu biblioteca CuPy este de 1.26 secunde, mai slab în anumite situații decât implementarea Numba.

Tabel 5. Determinarea numărului optim de fire de execuție - (920, 1600)– Filtru 7x7

Fire per Bloc / Blocuri per Grilă	Timp Numba
(16, 16) / (60, 100)	1.09 s
(16, 32) / (60, 50)	0.49 s
(32, 16) / (30, 100)	0.51 s
(32, 20) / (30, 80)	0.58 s
(20, 32) / (48, 50)	0.61 s
(32, 32) / (30, 50)	1.09 s

În cazul unei imaginii a cărei dimensiune nu mai depășește la fel de mult numărul de nuclee CUDA dispuse de placa grafică, se poate observa mai ușor influența numărului de fire de execuție lansate și a numărului de blocuri. În acest caz, în situațiile în care dimensiunile blocului de thread-uri se potrivește cu aspectul imaginii, calculul este mai eficient decât în cazul opus, iar în cazul în care dimensiunile sunt egale, performanțele sunt, de asemenea, mai slabe. În cazul acestei imagini de dimensiuni mult mai reduse decât cazul Tabel 4, aplicarea numărului maxim de thread-uri nu mai este la fel de eficient, numărul ideal fiind unul mai redus (16, 32). De această dată, biblioteca CuPy a fost mult mai eficientă decât Numba, obținând 0.31 secunde timp de rulare, în cel mai bun caz.



Figură 3.1. Imaginea originală și cea rezultată a Tabelului 5

În ultimul rând, este demn de menționat faptul că filtrarea întregii imagini satelitare cu ajutorul uneltei software SNAP Toolbox (en. abrev. SeNtinel Applications Platform), concepută de către ESA (en. abrev. European Space Agency), cu un filtru de aceeași dimensiune și caracteristici, durează 23 de secunde, comparativ cu cea mai rapidă metodă Numba ce durează doar 0.76 secunde. Accelerarea paralelă ajunge să fie de până la 30 de ori mai mare.

4. Concluzii

Întregul experiment a demonstrat, în esență, că utilizarea unei biblioteci cu accelerare GPU este mult mai eficientă decât utilizarea unei biblioteci clasice cu accelerare CPU, în scenariul filtrării imaginilor satelitare SAR de dimensiuni mari – milioane, zeci de milioane de pixeli. Chiar și în scenariul imaginilor ce conțin zeci sau sute de mii de pixeli, timpul de încărcare al Kernel-ului și al datelor ce sunt destinate prelucrării pe GPU este mult mai scurt decât întregul proces de prelucrare al imaginilor folosind arhitectura multinucleu a CPU.

Cu cât dimensiunea imaginii satelitare prelucrate este mai mare, cu atât accelerarea arhitecturii CUDA crește comparativ cu cea CPU. Performanța arhitecturii CUDA se menține constantă pe măsura creșterii dimensiunii imaginii prelucrate, pe când arhitectura CPU începe să fie depășită computațional, timpul de rulare crescând constant.

Iar cum filtrul Lee-Sigma este mult mai eficient în cazul imaginilor de rezoluție mai mare, filtrarea imaginilor de rezoluție mică fiind imprecisă, calitatea detaliilor crește semnificativ în urma filtrării unei imagini SAR de dimensiuni mari (> 1000x1000 pixeli). Astfel, ideea utilizării unei biblioteci cu accelerare CPU rămâne doar o ultimă soluție de urgență, atunci când nu se dispune de o placă grafică cu arhitectură CUDA.

Dimensiunea ferestrei de filtrare nu afectează timpul computațional total, ci doar rezultatul final al filtrării, în funcție de nivelul de detaliu ce se dorește a fi păstrat și dimensiunea imaginii filtrate.

Conform rezultatelor obținute în Capitolul 3, partea a doua, alegerea numărului de fire de execuție și al numărului de blocuri trebuie făcut în mod corelat față de dimensiunea datelor, tipul acestora, capacitatea maximă a hardware-ului și astfel încât cât mai multe din thread-urile generate să fie active. În mod ideal, ar fi bine ca dimensiunea datelor prelucrate să fie multiplu de dimensiunea warp-ului, însă acest lucru nu este posibil, iar când acest lucru se întâmplă, apar încetiniri din pricina thread-urilor generate suplimentar – cazul Tabel 1.

În final, deși biblioteca Numba este adesea puțin mai lentă decât CuPy, datorită simplității și a similarității modului de lucru cu biblioteca CUDA din C++, Numba este alternativa optimă pentru implementarea rapidă a funcțiilor formate din operații algebrice liniare direct în cod Python. Acest lucru permite adoptarea directă a funcțiilor de filtrare implementate, precum Lee-Sigma, în lanțul de prelucrare al imaginilor satelitare ce are loc în vederea pregătirii datelor pentru antrenarea unei rețele neurale adânci.

Bibliografie

[1] "Speckle (interference) - Wikipedia." Accessed: May 21, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Speckle_%28interference%29

[2] A. Braun and L. Veci, "Sentinel-1 Toolbox SAR Basics Tutorial," 2020, Accessed: May 21, 2024. [Online]. Available: http://step.esa.int

[3] "Speckle noise Filtering on Sentinel-1 Images (RADAR (SAR) images) in the SNAP - YouTube." Accessed: May 21, 2024. [Online]. Available: https://www.youtube.com/watch?v=oKy0IAxYuaI

[4] "Denoising Sentinel-1 Radar Images with Neural Networks | by Dheeptha Badrinarayanan | Medium." Accessed: May 21, 2024. [Online]. Available: https://medium.com/@dheeptha/denoising-sentinel-1-radar-images-5f764faffb3e

[5] "Cornell Virtual Workshop > Understanding GPU Architecture > GPU Characteristics > Design: GPU vs. CPU." Accessed: May 21, 2024. [Online]. Available: https://cvw.cac.cornell.edu/gpu-architecture/gpu-characteristics/design

[6] "Introduction to CUDA Programming - GeeksforGeeks." Accessed: May 21, 2024. [Online]. Available: https://www.geeksforgeeks.org/introduction-to-cuda-programming/

[7] "NVIDIA ® CUDA™ Architecture Introduction & Overview," 2009, Accessed: May 21, 2024. [Online]. Available: www.nvidia.com/cuda

[8] "CUDA-X GPU-Accelerated Libraries | NVIDIA Developer." Accessed: May 21, 2024. [Online]. Available: https://developer.nvidia.com/gpu-accelerated-libraries

[9] "| CUDA programming: grid of thread blocks (Source: NVIDIA). | Download Scientific Diagram." Accessed: May 21, 2024. [Online]. Available: https://www.researchgate.net/figure/CUDA-programming-grid-of-thread-blocks-Source-NVIDIA_fig3_328752788

[10] "Speckle function—ArcMap | Documentation." Accessed: May 21, 2024. [Online]. Available: https://desktop.arcgis.com/en/arcmap/latest/manage-data/raster-and-images/speckle-function.htm

[11] "Image Processing Class #4 — Filters | by Pitchaya Thipkham | Towards Data Science." Accessed: May 23, 2024. [Online]. Available: https://towardsdatascience.com/image-processing-class-egbe443-4-filters-aa1037676130

[12] "senbox-org/esa-snappy." Accessed: May 21, 2024. [Online]. Available: https://github.com/senbox-org/esa-snappy

[13] "Numba: A High Performance Python Compiler." Accessed: May 21, 2024. [Online]. Available: https://numba.pydata.org/

[14] "numba · PyPI." Accessed: May 21, 2024. [Online]. Available: https://pypi.org/project/numba/

[15] "Numba - Wikipedia." Accessed: May 21, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Numba

[16] “numba/numba: NumPy aware dynamic Python compiler using LLVM.” Accessed: May 21, 2024. [Online]. Available: https://github.com/numba/numba

[17] “CuPy: NumPy & SciPy for GPU.” Accessed: May 21, 2024. [Online]. Available: https://cupy.dev/

[18] “Python CuPy - GeeksforGeeks.” Accessed: May 21, 2024. [Online]. Available: https://www.geeksforgeeks.org/python-cupy/

[19] “Overview — CuPy 13.1.0 documentation.” Accessed: May 21, 2024. [Online]. Available: https://docs.cupy.dev/en/stable/overview.html

[20] “Schematic of the concept for the laser speckle investigation | Download Scientific Diagram.” Accessed: May 21, 2024. [Online]. Available: https://www.researchgate.net/figure/Schematic-of-the-concept-for-the-laser-speckle-investigation_fig1_374932663

Anexă

1. Implementarea pașilor inițiali de încărcare a imaginii și a bibliotecilor

```
import time
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as colors
import cupy as cp
import cupyx.scipy.ndimage
from numba import cuda


# incarcare img satelitara
img = np.load('band-Intensity_VV.npy')[1700:2500, 6600:7400]
img = np.ascontiguousarray(img) # pas aditional CUDA
    # toata imaginea in aceeasi zona de memorie
print(np.shape(img))
print(np.min(img))
print(np.max(img))
print(type(img))

# afisare imagine originala
plt.figure(), plt.imshow(img, cmap='gray', norm=colors.LogNorm()), plt.colorbar()
```

2. Inițializarea ferestrei de filtrare, măsurarea timpului, afișarea rezultatelor

```
# setare dimensiune filtru
filter_size = 3
capat = filter_size // 2

# aplicare si afisare filtru Lee-Sigma
start = time.time()
img_lee = apply_lee_sigma_filter(img, filter_size, 0.54, 1)
stop = time.time()
print(f'Timp total:{stop-start} sec')
plt.figure("LeeSigma"), plt.imshow(img_lee, cmap='gray', norm=colors.LogNorm()),
plt.colorbar(), plt.show()
```

3. Implementarea funcției de filtrarea – NumPy

```
def apply_lee_sigma_filter(img, size, Cu, Cmax):
    h, w = img.shape
    capat = size // 2
    new_img = np.zeros((h, w), dtype=np.float64)

    for i in range(capat, h - capat):
        for j in range(capat, w - capat):
            vec = img[i - capat:i + capat + 1, j - capat:j + capat + 1]
            new_img[i, j] = lee_sigma_filter(vec, size, Cu, Cmax)

    # Se ocupa de taierea marginilor pron copierea acestora (metoda simpla utilizata pentru
evitarea brodarii cu valori 0)
    new_img[:capat, :] = img[:capat, :]
    new_img[-capat:, :] = img[-capat:, :]
    new_img[:, :capat] = img[:, :capat]
    new_img[:, -capat:] = img[:, -capat:]

    return new_img
```

4. Implementarea filtrului Lee-Sigma – NumPy

```
def lee_sigma_filter(vec, size, Cu, Cmax):
    """
    Aplica filtrul Lee-Sigma pentru o fereastra locala de pixeli.

    Parameters:
    vec : 2D numpy array
        Vecinatatea de pixeli din jurul pixelului de interes
    size : int
        Dimensiunea ferestrei
    Cu : float
        Constanta universala pentru scalarea zgomotului, de obicei in jur de 0.523.
    Cmax : float
        Coeficientul maxim de varianta a zgomotului, definirea pragului de similaritate pentru
varianta.

    Returns:
    float
        Pixelii cu valori filtrate.
    """
    center = size // 2
    local_mean = np.mean(vec, dtype=np.float64)
    local_var = np.var(vec, dtype=np.float64)

    # Estimarea variantei zgomotului ca produs dintre constanta universala si media locala
    noise_var = Cu * local_mean

    # Calculul coeficientului sigma patrat (sigma^2)
    sigma_squared = local_var / (local_mean ** 2) if local_mean != 0 else 0

    if sigma_squared <= Cmax:
        weight = 1 - noise_var / (local_var if local_var != 0 else 1)  # Evitare impartire cu
0
        weight = np.clip(weight, 0, 1)  # Asigurarea valorii ponderii intre 0 si 1
    else:
        weight = 0

    # Calculul valorilor filtrate
    filtered_value = local_mean * weight + vec[center, center] * (1 - weight)
    return filtered_value
```

5. Implementarea funcției de filtrare – Numba:

```
    # Inițializarea și rularea kernelului
    def apply_lee_sigma_filter(img, size, Cu, sigma):
        # Prepară datele și configurația kernelului
        d_img = cuda.to_device(img)
        d_result = cuda.device_array_like(img)

        threadsperblock = (16, 16)
        blockspergrid_x = (img.shape[0] + (threadsperblock[0] - 1)) // threadsperblock[0]
        blockspergrid_y = (img.shape[1] + (threadsperblock[1] - 1)) // threadsperblock[1]
        blockspergrid = (blockspergrid_x, blockspergrid_y)

        # Lansează kernelul
        lee sigma filter kernel[blockspergrid, threadsperblock](d img, d result, size, Cu,
    sigma)

        # Copiază rezultatul înapoi pe CPU și returnează rezultatul
        result = d_result.copy_to_host()
        return result
```

6. Implementarea filtrului Lee-Sigma – Numba Kernel:

```
@cuda.jit
def lee_sigma_filter_kernel(d_img, d_result, size, Cu, sigma):
    x, y = cuda.grid(2)
    if x >= d_img.shape[0] or y >= d_img.shape[1]:
        return  # Verifică dacă indexul este în afara limitelor imaginii

    # Definirea ferestrei
    half_size = size // 2
    start_x = max(x - half_size, 0)
    end_x = min(x + half_size + 1, d_img.shape[0])
    start_y = max(y - half_size, 0)
    end_y = min(y + half_size + 1, d_img.shape[1])

    # Calculul mediei și variației locale
    local_sum = 0.0
    local_sum_sq = 0.0
    pixel_count = 0

    for i in range(start_x, end_x):
        for j in range(start_y, end_y):
            pixel_val = d_img[i, j]
            local_sum += pixel_val
            local_sum_sq += pixel_val ** 2
            pixel_count += 1

    if pixel_count > 0:
        local_mean = local_sum / pixel_count
        local_var = (local_sum_sq / pixel_count) - (local_mean ** 2)
        noise_var = Cu * local_mean

        # Calculul coeficientului de variație (sigma^2)
        if local_mean != 0:
            sigma_squared = local_var / (local_mean ** 2)
        else:
            sigma_squared = 0

        # Calculul ponderii
        if sigma_squared <= sigma:
            weight = 1 - noise_var / max(local_var, 1e-6)  # Evită împărțirea la zero
            weight = max(min(weight, 1), 0)  # Asigură că greutatea este între 0 și 1
        else:
            weight = 0

        # Aplicarea filtrului
        d_result[x, y] = local_mean * weight + d_img[x, y] * (1 - weight)
```

7. Implementarea funcției de filtrare și a filtrului Lee-Sigma – CuPy:

```
def gpu_lee_sigma_filter(img, size, Cu, Cmax):
    img_gpu = cp.asarray(img)  # Transfer image to GPU
    h, w = img_gpu.shape
    capat = size // 2

    # Prepare output image with the same type as input
    new_img_gpu = cp.zeros((h, w), dtype=img_gpu.dtype)

    # Define the kernel for averaging (uniform filter)
    kernel = cp.ones((size, size), dtype=cp.float32) / (size * size)

    # Apply padding to handle boundaries: reflect, constant, nearest, mirror, wrap
    img_padded = cp.pad(img_gpu, pad_width=capat, mode='reflect')

    # Compute local mean and local mean of squares using convolutions
    local_mean = cupyx.scipy.ndimage.convolve(img_padded, kernel, mode='reflect')[capat:-
capat, capat:-capat]
    local_sqr_mean = cupyx.scipy.ndimage.convolve(img_padded**2, kernel,
mode='reflect')[capat:-capat, capat:-capat]
    local_var = local_sqr_mean - local_mean**2

    # Calculate noise variance and coefficient of variation squared
    noise_var = Cu * local_mean
```

```
    sigma_squared = local_var / (local_mean ** 2 + 1e-10)  # Avoid division by zero

    # Calculate weights
    weight = cp.where(sigma_squared <= Cmax, 1 - noise_var / cp.maximum(local_var, 1e-10),
0)
    weight = cp.clip(weight, 0, 1)  # Ensure the weight is between 0 and 1

    # Compute the filtered image
    new_img_gpu = local_mean * weight + img_gpu * (1 - weight)

    # Transfer result back to CPU
    return cp.asnumpy(new_img_gpu)
```