

Universitatea POLITEHNICA din București
Facultatea de Electronică Telecomunicații, și Tehnologia
Informației

Implementarea și accelerarea FPGA ale unui sistem de
localizare a sursei de semnal radio

Proiect de diplomă

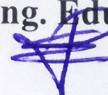
Prezentat ca cerință parțială pentru obținerea titlului de *Inginer*
în domeniul *Electronică și Telecomunicații*
programul de studii *Tehnologii și Sisteme de Telecomunicații*

Conducător științific
Ș.I. Dr. Ing. Lucian Petrică

Absolventă
Fluerătoru Laura-Ștefania

Universitatea "Politehnica" din București
Facultatea de Electronică, Telecomunicații și Tehnologia Informației
Departamentul TELECOMUNICAȚII

Aprobat Director de Departament:
Conf. Dr. Ing. ~~Eduard~~ Popovici



**IMPLEMENTAREA ȘI ACCELERAREA FPGA A UNUI SISTEM DE
LOCALIZARE A SURSEI DE SEMNAL RADIO
a studentei Fluerațoru I. Laura-Ștefania, Grupa 444C**

1. Titlul temei: **Implementarea și accelerarea FPGA a unui sistem de localizare a sursei de semnal radio**

2. Contribuția practică, originală a studentului va consta în (*în afara părții de documentare*):

Se va implementa un algoritm pentru estimarea locației unei surse de semnal radio pe un accelerator SIMD cu arhitectura ConnexArray, care este controlat de un procesor ARM, se va integra în cadrul GNU Radio, un framework pentru radio definit în program (SDR), se va testa funcționarea sistemului obținut și se vor evalua performanțele sale.

Se va pleca de la o implementare a unui algoritm de localizare a sursei radio pentru arhitectura x86. Apoi, codul va fi adaptat pentru procesorul SIMD folosit, ceea ce implică vectorizarea lui și utilizarea resurselor limitate de care dispune procesorul. Se vor folosi OPINCAA, un mediu de programare pentru procesoare SIMD care implementează arhitectura ConnexArray, și limbajele C++ și Python.

3. Proiectul se bazează pe cunoștințe dobândite în principal la următoarele 3-4 discipline: Arhitectura microprocesoarelor, Semnale și Sisteme, Prelucrarea Digitală a Semnalelor, Comunicații Analogice și Digitale

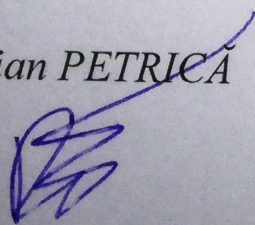
4. Proprietatea intelectuală asupra proiectului aparține: UPB/ studentului

5. Locul de desfășurare a activității: UPB

6. Data eliberării temei: 20 octombrie 2016

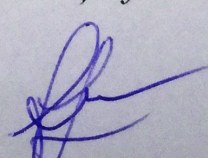
CONDUCĂTOR LUCRARE:

As. Prof. Lucian PETRICĂ



STUDENT:

Laura-Ștefania FLUERĂTORU



Declarație de onestitate academică

Prin prezenta declar că lucrarea cu titlul *Implementarea și accelerarea FPGA ale unui sistem de localizare a sursei de semnal radio*, prezentată în cadrul Facultății de Electronică, Telecomunicații și Tehnologia Informației a Universității "Politehnica" din București ca cerință parțială pentru obținerea titlului de *Inginer* în domeniul *Electronică și Telecomunicații*, programul de studii *Tehnologii și Sisteme de Telecomunicații* este scrisă de mine și nu a mai fost prezentată niciodată la o facultate sau instituție de învățământ superior din țară sau străinătate.

Declar că toate sursele utilizate, inclusiv cele de pe Internet, sunt indicate în lucrare, ca referințe bibliografice. Fragmentele de text din alte surse, reproduse exact, chiar și în traducere proprie din altă limbă, sunt scrise între ghilimele și fac referință la sursă. Reformularea în cuvinte proprii a textelor scrise de către alți autori face referință la sursă. Înțeleg că plagiatul constituie infracțiune și se sancționează conform legilor în vigoare.

Declar că toate rezultatele simulărilor, experimentelor și măsurărilor pe care le prezint ca fiind făcute de mine, precum și metodele prin care au fost obținute, sunt reale și provin din respectivele simulări, experimente și măsurători. Înțeleg că falsificarea datelor și rezultatelor constituie fraudă și se sancționează conform regulamentelor în vigoare.

București, Iulie 2017

Absolventă: Fluerătoru Laura-Ștefania



Cuprins

1	Introducere	15
2	Fundamentul teoretic	17
2.1	Rolul algoritmilor de localizare a sursei unui semnal	17
2.2	Algoritmi de localizare a sursei unui semnal	17
2.3	Algoritmul MUSIC	18
2.4	Arhitectura Connex-ARM	21
2.5	Radio definit software	23
2.6	GNU Radio	25
3	Implementarea software	27
3.1	Implementarea DoA folosită	27
3.2	Descrierea funcționalității	27
3.3	Evaluarea profilului de execuție al implementării algoritmului MUSIC	31
3.4	Testare	38
3.5	Concluzii	39
4	Accelerarea hardware	41
4.1	Aspecte specifice în implementarea kernelurilor	41
4.2	Înmulțirea vectorilor de numere complexe pe procesorul ConnexArray	43
4.3	Kernel pentru înmulțirea unui vector cu o matrice	45
4.4	Kernel pentru produsul înlănțuit dintre un vector linie, o matrice pătratică și un vector coloană	50
4.5	Kernel pentru autocorelația unui semnal	52
4.6	Integrarea kernelurilor în GNU Radio	54
5	Evaluarea performanțelor obținute	59
5.1	Metodologia de evaluare a performanțelor	59
5.2	Evaluarea performanțelor blocului ce calculează autocorelația semnalului de intrare	60
5.3	Evaluarea performanțelor blocului ce calculează spectrul MUSIC	63
5.4	Evaluarea performanțelor lanțului de procesare MUSIC folosind ConnexArray .	66
5.5	Concluzii referitoare la performanțele obținute	68
6	Concluzii	69
1	Testarea lanțului de procesare MUSIC	73
1.1	Declararea claselor necesare în testare	73
1.2	Definirea metodei de construcție a unui lanț de procesare	75
2	Kerneluri ConnexArray	79
2.1	Kernel pentru înmulțirea a doi vectori	79

2.2	Kernel pentru înmulțirea mai multor vectori cu aceeași matrice	80
2.3	Kernel pentru înmulțirea înlănțuită dintre un vector linie, o matrice și un vector coloană	81
2.4	Kernel pentru realizarea autocorelației	82
3	Evaluarea performanțelor kernelurilor pentru acceleratorul ConnexArray	85
3.1	Măsurarea timpului de execuție al unui program	85

Listă de figuri

2.1	Configurația inițială în algoritmul MUSIC	18
2.2	Arhitectura Connex-ARM [17]	22
2.3	Legătura dintre tehnologiile SDR, AR, CR și IR	24
3.1	Lanțul de procesare din implementarea algoritmului MUSIC folosită	28
3.2	Spectrul MUSIC	32
3.3	Unghiul de incidență estimat pentru unul dintre semnalele de intrare	33
3.4	Vizualizarea profilului de execuție pe un procesor Intel Core i7 cu ajutorul FlameGraph	36
4.1	Structura memoriei locale a acceleratorului ConnexArray	42
4.2	Înmulțirea a doi vectori de numere complexe pe ConnexArray	44
4.3	Exemplu de aranjare a elementelor în ConnexArray pentru o înmulțire dintre un vector și o matrice	46
4.4	Exemplu de aranjare a elementelor în ConnexArray pentru o înmulțire dintre doi vectori și aceeași matrice	48
4.5	Organizarea elementelor în memoria locală a acceleratorului în cazul înmulțirii dintre un vector linie și o matrice.	50
4.6	Aranjarea elementelor vectorilor în ConnexArray în cazul înmulțirii în lanțuite a unui vector, o matrice și a altui vector	52
4.7	Aranjarea datelor de intrare în acceleratorul ConnexArray pentru un kernel care calculează autocorelația unui semnal	54
4.8	Structura intrărilor și ieșirilor blocului GNU Radio care realizează înmulțirea dintre o serie de vectori de intrare și aceeași matrice	55
4.9	Reprezentările corectă și incorectă ale spectrului MUSIC	57
4.10	Structura intrărilor și ieșirilor blocului GNU Radio care realizează autocorelația	58
5.1	Modul de lucru distribuit al blocului de autocorelație	62
5.2	Timpul de execuție al blocului de autocorelație pe procesorul ARM pentru o captură de 2048 eșantioane	63
5.3	Timpul de execuție al blocului de autocorelație pe procesorul ARM pentru o captură de 1024 eșantioane	63
5.4	Timpul de execuție al blocului ce calculează spectrul MUSIC cu 1024 elemente	64
5.5	Timpul de execuție al lanțului de procesare MUSIC în cele trei variante pentru o dimensiune a spectrului MUSIC de 1024 de elemente	65
5.6	Timpul de execuție al lanțului de procesare MUSIC pentru o dimensiune a spectrului MUSIC de 1024 de elemente	67

Listă de tabele

3.1	Profilul lanțului de procesare MUSIC realizat pe un procesor Intel Core i7 . . .	35
3.2	Profilul lanțului de procesare MUSIC realizat pe un procesor Intel Core i7 după aplicarea corecțiilor în algoritmul de identificare a maximelor spectrului MUSIC	35
3.3	Profilul lanțului de procesare MUSIC realizat pe un procesor ARM Cortex A9 după aplicarea corecțiilor în algoritmul de identificare a maximelor spectrului MUSIC	37
5.1	Profil pentru blocul de autocorelație care folosește un kernel ConnexArray . . .	60
5.2	Profil pentru blocul de autocorelație în implementarea originală	61
5.3	Profil pentru blocul care calculează spectrul MUSIC cu un kernel ConnexArray .	64
5.4	Profil de execuție pentru lanțul MUSIC folosind kernelul de autocorelație	68
5.5	Profil de execuție pentru lanțul MUSIC ce folosește kernelul MUSIC	68

Lista acronimelor

ALU – Arithmetic Logical Unit
AR – Adaptive Radio
CR – Cognitive Radio
DDR – Double Data Rate
DMA – Direct Memory Access
DoA – Direction of Arrival
DSA – Dynamic Spectrum Access
ESPRIT – Estimation of Signal Parameters via Rotational Invariant Techniques
FPGA – Field Programmable Gate Array
GPP – General Purpose Processor
I/O – Input/Output
IR – Intelligent Radio
LS – Local Storage
MUSIC – Multiple Signal Classification
MVDR – Minimum Variance Distortionless Response
OOT – Out-Of-Tree
PE – Processing Elements
PL – Programmable Logic
PMU – Performance Monitoring Unit
PS – Processing System
SAD – Sum of Absolute Differences
SDMA – Space-Division Multiple Access
SDR – Software Defined Radio
SDRAM – Synchronous Dynamic Random-Access Memory
SIFT – Scale-Invariant Feature Transform
SIMD – Single Instruction Multiple Data
SoC – System-on-Chip
SSD – Sum of Squared Differences

Capitolul 1

Introducere

Sistemele radio definite software (SDR - Software Defined Radio), în care funcții din nivelul fizic al echipamentelor sunt implementate software, a prins amploare odată cu creșterea capabilității procesoarelor de uz general, astfel încât, în mod curent, toate echipamentele radio au implementat într-o oarecare măsură acest concept. Migrarea către software nu este deloc surprinzătoare: echipamentele devin mai ușor (re)configurabile, actualizările se efectuează rapid, avantaj deosebit de important în contextul evoluției rapide a tehnologiei, iar costurile scad, deoarece nu se mai impune înlocuirea unor componente *hardware* care nu mai corespund cerințelor pieței.

Deși procesoarele de uz general s-au adaptat cerințelor de procesare impuse de procesarea digitală a semnalelor prin implementarea unor extensii multimedia pentru instrucțiuni SIMD, acestea sunt limitate din punct de vedere al capacității oferite, iar compilatoarele adesea nu pot efectua o vectorizare eficientă a codului și în unele cazuri nu oferă deloc suport pentru instrucțiuni SIMD. Dorim să găsim o alternativă pentru procesarea digitală a semnalelor mai eficientă folosind suportul unui procesor, fără a sacrifica programabilitatea atât de importantă în SDR.

Propunem folosirea unui ansamblu procesor-accelerator cu suport pentru programarea acceleratorului în interiorul codului executat pe procesor, care va efectua funcții de control, lăsând în sarcina acceleratorului părțile intensive din punct de vedere computațional care ar putea beneficia de suport SIMD. Ansamblul reprezintă o cale de mijloc între resursele de procesare insuficiente oferite de către extensiile multimedia și programarea dificilă a sistemelor care decuplează total procesorul gazdă de un accelerator adaptat perfect cerințelor aplicației pentru care este folosit. Sistemul utilizat este denumit Connex-ARM [1] și este implementat pe platforma Xilinx Zynq-7000 [2], în al cărei chip FPGA se află acceleratorul cu arhitectura ConnexArray.

Vom analiza performanțele sistemului obținute în localizarea sursei unui semnal radio, în care anumite puncte critice din lanțul de procesare vor fi implementate pe accelerator. Algoritmul folosit pentru localizarea sursei unui semnal este MUSIC [3] (MUltiple SIgnal Classification), iar implementarea folosită ca referință [4] este integrată în mediul GNU Radio [5], dedicat aplicațiilor SDR. Ansamblul folosit prezintă, în plus, o alternativă eficientă din punct de vedere energetic [6], aspect deosebit de important în echipamentele de telecomunicații.

Proiectul cuprinde, astfel, următoarele etape:

- Studiul algoritmului folosit și al tehnologiilor folosite în proiect, a căror funcționare va fi

sintetizată în Capitolul 2.

- Identificarea punctelor critice din lanțul de procesare care implementează algoritmul MUSIC, care va fi făcută în Capitolul 3, alături de o descriere a funcționării lanțului de procesare și a metodologiei utilizate în realizarea profilului său de execuție.
- Implementarea unor *kerneluri* pentru acceleratorul ConnexArray corespunzătoare punctelor critice identificate, detaliate în Capitolul 4. În acest capitol se vor descrie și pașii efectuați în integrarea kernelurilor în mediul GNU Radio, integrate în blocuri de sine stătătoare corespunzătoare unei funcționalități din lanțul MUSIC.
- Evaluarea performanțelor blocurilor nou create în comparație cu cele inițiale atât separat, cât și în întregul lanț de procesare va fi descrisă în Capitolul 5.

În finalul lucrării, în Capitolul 6, vom prezenta concluziile proiectului, precum și diverse îmbunătățiri care îi vor fi aduse pe viitor.

Capitolul 2

Fundamentul teoretic

2.1 Rolul algoritmilor de localizare a sursei unui semnal

De-a lungul anilor, cerințele impuse asupra sistemelor de telecomunicații wireless au crescut radical, devenind din ce în ce mai importantă eficiența lor spectrală. Rețelele trebuie să facă față unui număr ridicat de utilizatori, unor aplicații multimedia cu cerințe mari de date, menținând, în același timp, o latență cât mai redusă. O tehnică deosebit de utilă care s-a impus pentru satisfacerea acestor exigențe este tehnica accesului multiplu cu diviziune în spațiu (SDMA), care permite alocarea aceluiași canal unor utilizatori din zone diferite, putând astfel reutiliza canale în interiorul unei celule.

Pe baza tehnicii SDMA s-au dezvoltat sisteme de antene inteligente, care se folosesc de algoritmi de procesare a semnalelor pentru a-și optimiza caracteristica de radiație în funcție de semnalele primite. Un rol important îl au algoritmi de localizare a sursei unui semnal, pe baza cărora, folosind un algoritm de beamforming adaptiv, antenele își ajustează amplitudinile și fazele pentru a transmite în direcția utilizatorului dorit. Astfel, crește capacitatea de acoperire în interiorul unei celule, iar sistemul este capabil să diferențieze semnalele utile de cele nedorite, ceea ce duce și la creșterea capacității sale.

2.2 Algoritmi de localizare a sursei unui semnal

Metodele folosite pentru estimarea direcției de incidență se clasifică, în funcție de criteriul folosit, în metode convenționale, metode bazate pe separarea subspațiilor zgomotului sau semnalului, metoda similitudinii maxime etc. Din prima categorie fac parte Metoda convențională de Beamforming, cunoscută și drept Metoda Bartlett, precum și Metoda Variantei Minime a lui Capon, denumită și MVDR. Ce au în comun aceste metode este faptul că formează un fascicul convențional, pe care îl scanează peste o anumită zonă, iar direcția care produce cea mai mare putere de ieșire este considerată direcția de incidență [7]. În practică, pentru metoda Bartlett, se scanează cu un anumit unghi θ aria dorită, obținându-se ca răspuns vectorul director $a(\theta)$. Pe baza acestuia, se calculează matricea de covarianță R_{xx} și puterea de ieșire se calculează conform Ecuației (2.1).

$$P(\theta) = \frac{a^H(\theta)R_{xx}a(\theta)}{a^H(\theta)a(\theta)} \quad (2.1)$$

Se găsește direcția de incidență ca fiind unghiul corespunzător valorii maxime a puterii de ieșire. Pentru a rezolva problemele de rezoluție slabă, se folosește un filtru liniar cu anumite ponderi pentru a evita distorsionarea semnalului de interes și se calculează, în schimb, inversa produsului dintre matricea de covarianță și vectorul director [8].

Metodele bazate pe separarea subspațiilor se folosesc de ortogonalitatea dintre subspațiul format de semnalul de intrare și subspațiul format de zgomotul suprapus peste acesta. Ele ating o rezoluție ridicată, dar cu cerințe mari de procesare și de stocare și depind de o corelație cât mai slabă între semnalul de intrare și zgomot. Dacă raportul semnal-zgomot este foarte scăzut, sau sursele sunt foarte apropiate, performanțele acestor algoritmi se deteriorează [9]. Acestei categorii îi aparțin algoritmi MUSIC și ESPRIT.

În această lucrare, am analizat algoritmul MUSIC atât datorită performanțelor sale, precum și modalității sale de analiză și estimare, care se pliază pe cerințele actuale din domeniul comunicațiilor, ce vor fi detaliate în continuare.

2.3 Algoritmul MUSIC

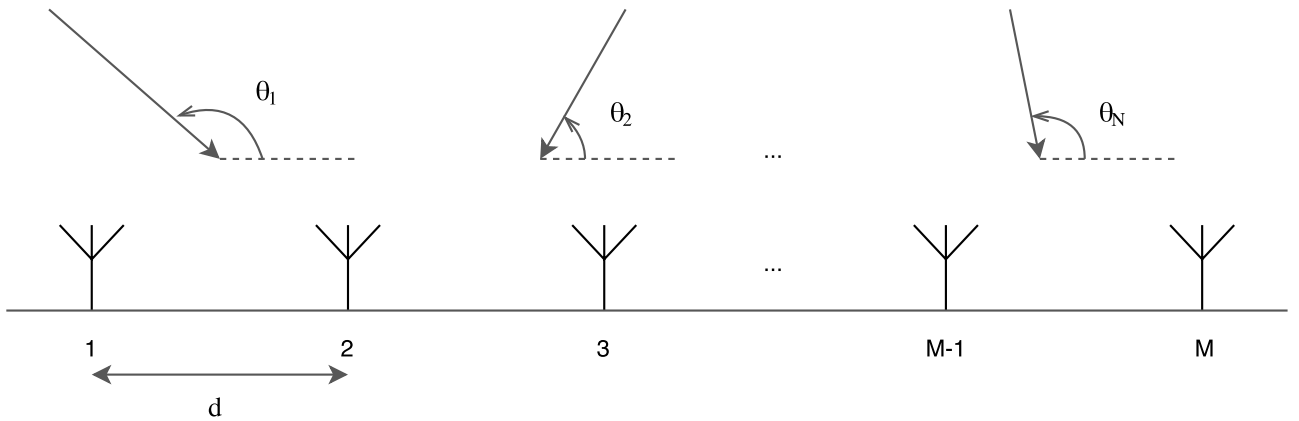


Figura 2.1: Configurația inițială în algoritmul MUSIC

În analiza algoritmului MUSIC presupunem că plecăm de la un șir liniar de M antene la care ajung N semnale $s_j(t)$, $i = \overline{1, N}$ și se dorește estimarea unghiului de incidență θ_j măsurat față de axa x în sens trigonometric sub care ajunge fiecare semnal la șirul de antene, ca în Figura 2.1. Pe parcursul lucrării, vom considera doar cazul unui sistem liniar de antene, deci ne vom referi la el pur și simplu ca la un „sistem de antene”, fără a mai menționa caracterul său liniar. Presupunem că mediul de propagare nu afectează semnificativ semnalele când acestea se propagă de la un element din șir la altul, deci semnalul care ajunge la o antenă diferă de cel care ajunge la o altă antenă doar printr-o întârziere τ .

Considerăm că prima antenă se află în originea sistemului, la locația $(0, 0)$ și exprimăm întârzierea semnalului de la celelalte elemente din șir relativ la semnalul care ajunge la elementul de referință. Astfel, pentru un șir liniar, semnalul j care ajunge la elementul $i = 2$ parcurge o distanță mai lungă cu $d \cos \theta_j$ față de primul element și, în cazul general, putem scrie întârzierea τ_i la elementul i ca

$$\tau_i = \beta(i - 1)d \cos \theta_j \quad (2.2)$$

unde $\beta = \frac{2\pi}{\lambda}$ este factorul de defazaj.

Aici, am presupus că defazajul semnalului depinde doar de locațiile distincte ale elementelor șirului de antene. În realitate, antenele vor avea și ele un răspuns dependent de directivitatea lor și de frecvență, care poate fi modelat ca un câștig g_i . Obținem, astfel, vectorul director (*steering vector*) pentru un anumit unghi de incidență θ_j și frecvență ω :

$$\mathbf{a}(\omega, \theta_j) = \begin{bmatrix} g_1(\omega, \theta_j)e^{j\tau_1(\theta_j)} \\ \dots \\ g_M(\omega, \theta_j)e^{j\tau_M(\theta_j)} \end{bmatrix} \quad (2.3)$$

Prin urmare, vectorul director este dependent de răspunsul individual al fiecărui element din șirul de antene, de geometria șirului, de frecvența semnalului și de unghiul de incidență al acestuia. Matricea care se formează din vectorii coloană directori pentru toate unghiurile de incidență și toate frecvențele se numește matricea colectoare a șirului (*array manifold matrix*).

$$\mathbf{A} = [\mathbf{a}(\omega, \theta_1) \quad \dots \quad \mathbf{a}(\omega, \theta_N)] \quad (2.4)$$

Dacă banda semnalului este suficient de îngustă, se poate considera, cu aproximație, că vectorul director este independent de frecvență și că depinde doar de unghiul de incidență. Mai mult, dacă presupunem că elementele șirului sunt izotrope, putem elimina dependența vectorului director de câștigul $g_i, i = \overline{1, M}$. Vom continua analiza având în vedere aceste presupuneri.

Putem scrie următoarea relație pentru semnalele de la intrarea fiecărei antene din șir:

$$\mathbf{x}(t) = [\mathbf{a}(\theta_1) \quad \mathbf{a}(\theta_2) \quad \dots \quad \mathbf{a}(\theta_N)] \begin{bmatrix} s_1(t) \\ s_2(t) \\ \dots \\ s_N(t) \end{bmatrix} + \begin{bmatrix} n_1(t) \\ n_2(t) \\ \dots \\ n_N(t) \end{bmatrix} \quad (2.5)$$

$$\mathbf{x}(t) = \mathbf{A}\mathbf{s}(t) + \mathbf{n}(t) \quad (2.6)$$

Algoritmul MUSIC (MUltiple Signal Classification) [3] face parte dintr-o clasă mai mare de algoritmi care se bazează pe metoda subspațiilor, care ia în considerare și zgomotul dintr-un sistem. Algoritmul oferă, în primul rând, informații despre numărul de semnale care ajung la un șir de antene și unghiul de incidență al acestora. Este un algoritm cu rezoluție mare, ceea ce înseamnă că poate distinge mai ușor decât alți algoritmi două semnale care vin din direcții foarte apropiate, dar are nevoie de o calibrare foarte precisă a șirului de antene. Calibrarea constă în obținerea matricei colectoare a șirului de antene; în practică, acest lucru se realizează măsurând răspunsurilor unor surse punctiforme ale șirului la diverse unghiuri și frecvențe. Pentru explicarea fundamentului matematic din spatele algoritmului MUSIC s-a folosit ca referință lucrarea [10], în care este tratat pe larg subiectul estimării unghiurilor de incidență folosind șiruri de antene.

În ecuația (2.5), vectorul \mathbf{x} al semnalelor de la intrarea antenelor din șir și vectorii directori $\mathbf{a}(\theta_j), j = \overline{1, N}$ pot fi priviți ca vectori într-un spațiu cu M dimensiuni, ceea ce înseamnă că \mathbf{x} poate fi scris ca o combinație liniară între vectorii directori, unde $s_j, j = \overline{1, N}$ sunt coeficienții combinațiilor.

Se poate calcula matricea de covarianță a intrării

$$\mathbf{R}_{xx} = E[\mathbf{x}\mathbf{x}^H] = \mathbf{A}E[\mathbf{s}\mathbf{s}^H]\mathbf{A}^H + E[\mathbf{n}\mathbf{n}^H] \quad (2.7)$$

$$\mathbf{R}_{xx} = \mathbf{A}\mathbf{R}_{ss}\mathbf{A}^H + \sigma_{zg}^2\mathbf{I} \quad (2.8)$$

S-a notat $\mathbf{R}_{ss} = E[\mathbf{s}\mathbf{s}^H]$ matricea de corelație a semnalului s . Se observă două proprietăți importante:

- Vectorii directori sunt liniar independenți, ceea ce înseamnă că matricea \mathbf{A} este de rang maxim.
- \mathbf{R}_{ss} este o matrice nesingulară dacă semnalele incidente sunt cel mult parțial necorelate. Dacă ar fi corelate, atunci cel puțin una dintre liniile/coloanele sale ar putea fi scrisă ca o combinație liniară a altor linii/coloane, ceea ce ar însemna că matricea ar avea determinantul egal cu 0, deci ar fi singulară și nu am mai putea folosi algoritmul.

Din aceste două proprietăți rezultă că, dacă $M < N$, atunci matricea $\mathbf{A}\mathbf{R}_{ss}\mathbf{A}^H$ este pozitiv semidefinită, având rangul N . Din această proprietate, se poate demonstra faptul că $M - N$ dintre valorile proprii ale matricei trebuie să fie nule. Folosind relația (2.6), reiese că atunci când $M - N$ dintre valorile proprii ale matricei $\mathbf{A}\mathbf{R}_{ss}\mathbf{A}^H$ sunt nule, cele mai mici valori proprii ale matricei \mathbf{R}_{xx} vor fi egale cu puterea zgomotului σ_{zg}^2 . Dacă notăm $\lambda_i, i = \overline{1, M}$ valorile proprii ale matricei \mathbf{R}_{xx} , atunci

$$\lambda_{N+1} = \lambda_{N+2} = \dots = \lambda_M = \lambda_{min} = \sigma_{zg}^2 \quad (2.9)$$

În realitate, însă, nu se va îndeplini egalitatea, deoarece folosim pentru estimare doar un număr finit de eșantioane, dar valorile vor fi, într-adevăr, foarte apropiate. Dacă notăm K multiplicitatea celei mai mici valori proprii a matricei \mathbf{R}_{xx} , atunci, știind că $M = N + K$, putem estima numărul semnalelor care ajung la șirul de antene

$$\hat{N} = M - K \quad (2.10)$$

Din definițiile valorilor proprii și a vectorilor proprii [11], vectorii proprii $\mathbf{v}_i, i = \overline{1, M}$ corespunzător celor mai mici valori proprii trebuie să satisfacă egalitatea

$$\mathbf{R}_{xx}\mathbf{v}_i = \sigma_{zg}^2\mathbf{v}_i, \quad i = \overline{N+1, M} \quad (2.11)$$

Folosind ecuația (2.6), înseamnă că

$$\mathbf{A}\mathbf{R}_{ss}\mathbf{A}^H\mathbf{v}_i = 0, \quad i = \overline{N+1, M} \quad (2.12)$$

Știind că \mathbf{A} este de rang maxim și că \mathbf{R}_{ss} este nesingulară, atunci

$$\mathbf{A}^H\mathbf{v}_i = 0, \quad i = \overline{N+1, M} \quad (2.13)$$

ceea ce este echivalent cu a spune că vectorii coloană ai matricei \mathbf{A} , adică vectorii directori, sunt perpendiculari pe vectorii proprii ai matricei \mathbf{R}_{xx} .

$$[\mathbf{a}(\theta_1) \quad \mathbf{a}(\theta_2) \quad \dots \quad \mathbf{a}(\theta_N)] \perp [\mathbf{v}_{N+1} \quad \mathbf{v}_{N+2} \quad \dots \quad \mathbf{v}_M] \quad (2.14)$$

Recapitulând, avem două subspații ortogonale: cel al semnalelor și cel al zgomotului. Vectorii directori ai șirului de antene aparțin subspațiului semnalelor și, așadar, sunt perpendiculari pe

subspațiul zgomotului, iar vectorii proprii ai matricei de covarianță \mathbf{R}_{xx} pot aparține oricăruia dintre cele două subspații. Prin urmare, putem să căutăm printre vectorii directori pe aceia care sunt ortogonali pe subspațiul zgomotului, în care se vor afla o parte din vectorii proprii ai matricei de covarianță, și să determinăm unghiurile de incidență θ_j .

Se calculează spectrul MUSIC folosind una dintre următoarele două formule:

$$P_{MUSIC}(\theta) = \frac{\mathbf{a}^H(\theta)\mathbf{a}(\theta)}{\mathbf{a}^H(\theta)\mathbf{V}_N\mathbf{V}_N^H\mathbf{a}(\theta)} \quad (2.15)$$

$$P_{MUSIC}(\theta) = \frac{1}{\mathbf{a}^H(\theta)\mathbf{V}_N\mathbf{V}_N^H\mathbf{a}(\theta)} \quad (2.16)$$

$$\mathbf{V}_N = [v_{N+1}, \dots, v_M] \quad (2.17)$$

și, în cazul în care vectorii directori sunt perpendiculari pe vectorii proprii, numitorul va fi minim, ceea ce va conduce la apariția unor vârfuri în spectru. Cunoaștem numărul de semnale estimat \hat{N} , deci cele \hat{N} vârfuri din spectru corespund unghiurilor de incidență căutate.

2.4 Arhitectura Connex-ARM

În această lucrare vor fi evaluate performanțele algoritmului studiat pe un sistem cu arhitectura Connex-ARM [1], ce conține elemente specifice atât paradigmei setului de instrucțiuni extins, cât și paradigmei procesor gazdă-accelerator (*host-accelerator paradigm*), și care este implementat pe platforma Xilinx Zynq-7000 [2]. Ambele abordări vizează procesări de tip Single Instruction Multiple Data (SIMD), care presupune efectuarea aceleiași operații în mod simultan pe o colecție de date pe un computer cu mai multe elemente de procesare (PEs).

Prima variantă, cea a unui set extins de instrucțiuni, beneficiază de o coordonare strânsă între extensia multimedia și procesorul de uz general (GPP), care, de cele mai multe ori, necesită un singur compilator pentru programul principal care se execută pe procesor și cu suport inclus pentru instrucțiunile SIMD. Din această categorie fac parte seturile de instrucțiuni Intel SSE/AVX [12] sau NEON [13] (disponibile pentru procesorul ARM Cortex-A9 de pe placa de dezvoltare utilizată), dar care sunt limitate în ceea ce privește capacitatea liniilor SIMD și gradul de paralelism atins.

În paradigma procesor gazdă-accelerator, funcțiile de control sunt executate pe GPP, iar sarcinile de lucru intensive din punct de vedere computațional sunt executate pe acceleratorare *multi-core*. În acest caz, există o libertate mai mare în privința arhitecturii acceleratorului, care poate fi adaptată în funcție de tipul de aplicații pentru care sistemul este destinat, cu care se pot atinge performanțe înalte. Dezavantajele acestei abordări constă în faptul că timpul de acces la accelerator poate crește considerabil și programarea lui este mai dificilă, având nevoie, în multe situații, de un anumit mediu de programare, precum NVIDIA CUDA [14] sau OpenCL [15].

Arhitectura Connex-ARM îmbină elemente caracteristice ambelor variante prezentate; ea este constituită dintr-un accelerator SIMD bazat pe procesorul multimedia Connex [16], denumit ConnexArray, implementat pe chipul FPGA din placa de dezvoltare, care se conectează la procesorul principal de tip ARM Cortex-A9 printr-o magistrală de date, ca în Figura 2.2. Cuplajul

dintre accelerator și procesor este suficient de strâns pentru a permite întârzieri relativ reduse ale accesului la date, dar necesită, totuși, un mediu de programare specific datorită faptului că este extern procesorului de uz general. Mediul de programare folosit în această arhitectură este OPINCAA [17] (Opcode Injection and Control for Accelerator Architectures), care permite includerea codului pentru accelerator în codul pentru GPP, cel dintâi putând fi trimis către accelerator în timpul execuției programului prin intermediul magistralei de date.

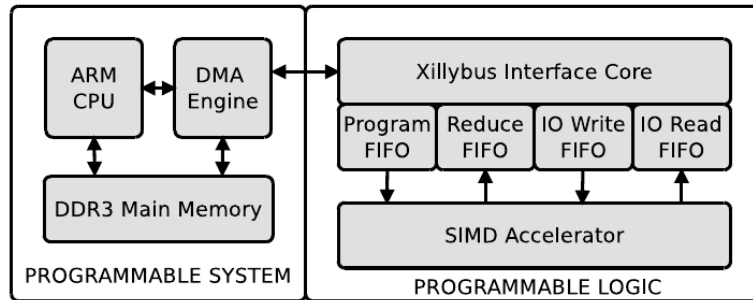


Figura 2.2: Arhitectura Connex-ARM [17]

Acceleratorul SIMD utilizat este format din N PEs, iar fiecare dintre acestea este echipat cu un set de registre interne, o unitate aritmetică logică (ALU), o logică internă de decodare a instrucțiunilor și o memorie locală (LS). La un moment de timp, în toate PEs se încarcă aceeași instrucțiune, putând selecta PEs pe care aceasta să se execute.

Schimbul de date dintre accelerator și procesorul ARM se face prin trei rețele corespunzătoare a patru cozi de preluare a datelor, după cum urmează [6]:

- Rețeaua de distribuție, corespunzătoare cozii `distributionFIFO`, este coada prin care instrucțiunile componente ale kernelului sunt distribuite către elementele de procesare (PEs) sub forma unui *fully pipelined logarithmic tree*.
- Rețeaua de I/O, corespunzătoare cozilor `writeFIFO` și `readFIFO`, controlează transferul de date de intrare și de ieșire dintre memoria principală a procesorului gazdă și memoria locală a acceleratorului (*local storage*).
- Rețeaua de reducere, corespunzătoare cozii `reductionFIFO`, care colectează operații globale precum reducerea unei sume, în care se adună toate elementele dintr-un anumit registru al elementelor de procesare sub forma unui *fully pipelined adder tree*.

În mediul de programare OPINCAA, codul pentru accelerator poate fi scris în C++, utilizând o sintaxă specifică, și se poate intercala cu codul scris pentru GPP. Se pot folosi date de tip vectorial și operatori caracteristici operațiilor disponibile pe accelerator, precum reducere, selecție de celule și operații aritmetice. Codul pentru accelerator este organizat în *kerneluri*, care sunt compilate și stocate până când se cere lansarea acestora în execuție.

În varianta utilizată în această lucrare, sistemul este implementat pe platforma Xilinx Zynq-7000, care este formată din două componente, identificabile și în Figura 2.2:

- **Sistemul de procesare (PS)**, în care sunt incluse un procesor ARM Cortex-A9, care lucrează la o frecvență de 667 MHz, un controller de memorie DDR SDRAM și un sistem de acces direct la memorie (DMA).

- **Logica programabilă (PL)**, alcătuită dintr-un FPGA Artix-7. PL se conectează la PS printr-o magistrală de date AMBA AXI, pentru implementarea de circuite digitale care să accelereze programele executate de GPP.

Arhitectura ConnexArray utilizată pune la dispoziție 128 PEs, operanți pe 16 biți, 32 registre și o memorie locală cu o capacitate de 256 KB și consumă, în medie, 600 mW, conform estimărilor efectuate cu programele puse la dispoziție de Xilinx [6]. Procesorul ARM dual-core consumă, potrivit documentației Zedboard, maxim 1.25 W, ajungând, în medie, la un total de aproximativ 2 W consumați de întregul sistem.

Ansamblul format oferă, prin urmare, o opțiune atractivă pentru procesări dedicate intensive din punct de vedere computațional, cu un consum de putere redus, fără a sacrifica programabilitatea. În analiza de imagini, de exemplu, pentru calculul normelor SAD (*sum of absolute differences*) și SSD (*sum of squared differences*) necesare în algoritmul SIFT (*scale-invariant feature transform*), s-a obținut un *throughput* de 4 - 6 ori mai bun folosind acest ansamblu, în comparație cu execuția folosind doar procesorul ARM, cu un consum de energie de trei ori mai mic, iar în comparație cu procesoare precum Intel Core i7 2600K sau NVidia GTX680 GPU are un consum de energie cu până la 40% mai scăzut. [6].

2.5 Radio definit software

Este considerat un dispozitiv radio, sau, pe scurt, un radio, un dispozitiv care transmite și recepționează semnale din domeniul radio al spectrului electromagnetic în scopul transmiterii informației. Un radio definit software, prescurtat și SDR, pornind de la denumirea în limba engleză Software Defined Radio, este „un radio în care o parte dintre sau toate funcțiile de nivel fizic sunt definite software” [18].

Deși conceptul de SDR era folosit de mai mult timp, până în 1992, odată cu publicația lui Joseph Mitola despre acest concept în IEEE [19], nu exista o viziune unificată asupra sa. Odată cu această publicație, subiectul a ajuns în atenția unui public mai larg și au fost, totodată, propuse mai multe direcții de cercetare care au contribuit la consolidarea domeniului.

Avantajul implementării software a funcționalităților de nivel fizic devine extrem de important în contextul în care tehnologia avansează rapid și se impune modificarea facilă a dispozitivelor comercializate. În plus, costurile scad deoarece, în majoritatea cazurilor, nu mai este necesară înlocuirea echipamentelor, ci doar actualizarea programelor care le descriu funcționarea. Erorile care pot apărea în anumite echipamente nu mai prezintă un risc major, deoarece și ele pot fi rezolvate prin actualizarea programului executat de echipament. Mai mult, odată definite anumite funcționalități, ele pot fi adaptate pentru mai multe produse, cu modificări minime dependente de platforma pe care se lucrează.

Conceptul a început să ia amploare încă de la sfârșitul anilor '90, când cele mai multe echipamente de pe piață foloseau un procesor de uz general (GPP) pentru funcții de nivel înalt în rețea precum interfațarea cu operatorul rețelei sau semnalizare și un procesor de semnal pentru modulație sau procesare digitală de semnale [20]. Acest lucru a fost văzut ca o oportunitate pentru extinderea funcționalității software a echipamentelor, astfel încât s-a căutat o adaptare mai amplă a lor la schimbările de mediu și la cerințele utilizatorilor. S-a pus problema unei

utilizări mai eficiente și dinamice a resurselor spectrale, alocarea mai inteligentă a canalelor, precum și relaxarea constrângerilor legate de interferențe, atunci când este posibil. Astfel, s-au concretizat domenii precum radio adaptiv, radio cognitiv și radio inteligent, fiecare o extindere a celui precedent, a căror implementare devine posibilă cu ajutorul SDR. Figura 2.3 sugerează relația dintre tehnologiile menționate.

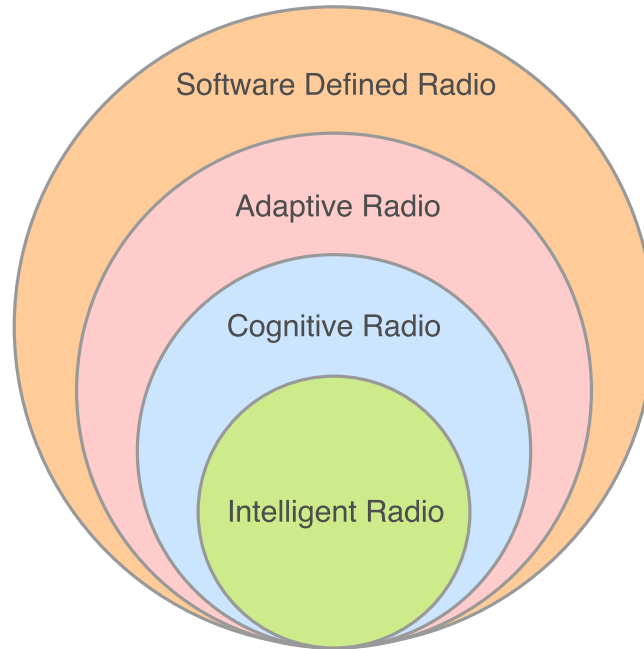


Figura 2.3: Legătura dintre tehnologiile SDR, AR, CR și IR

Deși nu există o definiție unanim acceptată a unui **radio adaptiv** (AR), se consideră că acesta este capabil să își monitorizeze performanțele și să reacționeze la variația acestora prin modificarea parametrilor săi de operare. În general, el folosește benzi de frecvență nelicențiate care nu sunt ocupate într-o anumită arie.

Un **radio cognitiv** (CR) merge mai departe, sistemul de comunicații devenind *conștient* de starea lui internă. El va putea, de exemplu, să recunoască activitățile utilizatorului și să îl asiste în îndeplinirea acestora, să reacționeze în funcție de locația curentă, recunoscând servicii disponibile într-o anumită arie. În partea de *backend*, invizibilă pentru utilizator, va putea să optimizeze nivelul rețea folosind metrici de performanță, să balanseze traficul și va avea un motor care cunoaște politicile zonelor geografice de operare ale echipamentelor și le respectă în funcție de cea în care lucrează. De departe, cele mai importante funcționalități sunt legate de utilizarea spectrului, de aici fiind desprins un domeniu separat - **accesul dinamic la spectru** (DSA). El presupune, printre altele, posibilitatea de a selecta banda de frecvență în care să lucreze echipamentul pentru a optimiza folosirea resurselor spectrale și a evita interferențele cu alte echipamente din jurul său. Cu ajutorul acestor tehnici, va putea să suporte un număr crescut de utilizatori, să fie conștient de spectrul disponibil, de interferențele și căile de propagare multiple din jurul său și de puterea semnalului recepțional, utilizând aceste informații pentru a crește robustețea serviciului [20].

Un **radio inteligent** (IR) este deja capabil de învățare automată. El nu numai că va reacționa în funcție de schimbările mediului, dar va învăța din trecut pentru a-și îmbunătăți performanțele.

În concluzie, viitorul sistemelor de comunicații prevede schimbări ambițioase, a căror limită va fi, în primul rând, puterea de procesare disponibilă pentru implementarea funcționalităților descrise, care intervine în raportul dintre costul necesar producerii și întreținerii unui echipament și beneficiile pe care acesta le aduce.

2.6 GNU Radio

GNU Radio [5] este un set de instrumente gratuit și cu sursă deschisă (*open source*) care facilitează implementarea blocurilor de procesare de semnal în dispozitivele radio definite software. El pune la dispoziție mai multe blocuri de procesare clasice, precum filtre, egalizatoare, modulatori, codare etc., și se ocupă de interconectarea blocurilor și schimbul de date dintre ele. De asemenea, conține și metode pentru definirea unor noi blocuri de procesare corespunzătoare nevoilor particulare ale dezvoltatorilor. El poate fi folosit atât pentru a realiza simulări pentru verificarea și stabilirea performanțelor anumitor blocuri, cât și pentru a se interfața cu dispozitive hardware externe pentru implementarea propriu-zisă a echipamentelor.

Avantajul său major este faptul că dezvoltatorul se poate concentra pe implementarea blocurilor pentru anumite funcționalități ale echipamentului, fără a fi nevoie să se ocupe și de comunicația dintre acestea, care este un alt domeniu în sine, ce necesită un organizator (*scheduler*) eficient ce distribuie sarcinile blocurilor existente în lanțul de comunicație. Acest lucru îl face să lucreze foarte bine atât pe arhitecturi puternice, cu mai multe nuclee, într-un mod de lucru distribuit, cât și pe procesoare *embedded* unde resursele de procesare sunt mai limitate.

Întrucât algoritmi de localizare a sursei unui semnal sunt folosiți cu precădere în echipamente de comunicații care se adaptează la mediul înconjurător, am considerat utilă integrarea kernelurilor pentru accelerarea algoritmului MUSIC în interiorul GNU Radio, pentru a le asigura disponibilitatea și re folosirea în mai multe situații. Faptul că implementarea sa software este disponibilă gratuit și *open source*, îl face ideal pentru lucrările din mediul academic. În acest context, este necesară descrierea unor concepte specifice radiourilor definite software folosite în GNU Radio și a modului său de lucru.

GNU Radio folosește conceptul de lanț de procesare (*flowgraph*) în organizarea unei aplicații. În acest mod, se face o separare între funcționalitățile ei, împărțindu-le în blocuri de procesare, care vor fi apoi interconectate. Fiecare lanț de procesare trebuie să conțină cel puțin o sursă și un bloc de „scurgere” a semnalului (*sink*). Între aceste două blocuri are loc procesarea propriu-zisă a semnalului. Pentru implementarea lor este posibilă utilizarea a două limbaje de programare: Python și C++, pentru acest proiect alegând cea de-a doua variantă, întrucât suntem interesați în primul rând de performanța modulelor și OPINCAA, mediul de programare pentru acceleratorul ConnexArray, oferă suport doar pentru acest limbaj de programare.

După cum am menționat deja, GNU Radio dispune de o bibliotecă unde sunt deja implementate o serie de module de uz general. Pentru implementarea unor funcționalități particulare, se pot defini module proprii, cunoscute și sub numele de module *out-of-tree* (OOT), cu ajutorul utilitarului `gr_modtool`, care va crea o structură de directoare specifică unui modul GNU Radio. Tot cu ajutorul acestui utilitar se pot adăuga și noi blocuri în modulul OOT nou creat, specificând anumite caracteristici ale acestuia: tipul, parametrii de intrare, limbajul folosit etc.

La crearea unui bloc, trebuie să analizăm relația dintre intrările și ieșirile sale, pentru a decide tipul acestuia. Printre tipurile standard de blocuri se numără `general`, `sync_block`, `interpolator` și `decimator`.

- `general` - nu presupune o relație anume între intrări și ieșiri, iar o eventuală stabilire a unei legături se poate face prin supraîncărcarea funcției `forecast`, care este apelată periodic pentru a stabili câte date de ieșire se pot produce, reținute în variabila `noutput_items`. În mod implicit, relația dintre intrări și ieșiri este de 1:1. În acest caz, blocul are o metodă pur virtuală denumită `general_work`, deci care trebuie supraîncărcată, în care se va efectua procesarea propriu-zisă din bloc. La sfârșitul ei, trebuie specificate câte date de intrare au fost consumate și să returnăm numărul de date de ieșire produse.
- `sync` - un bloc care consumă și produce un număr egal de date pe fiecare port. Funcția `forecast` este, în acest caz, predefinită și metoda în care are loc procesarea poartă denumirea `work`.
- `source` și `sink` - blocuri care au doar ieșiri, respectiv doar intrări.
- `decimator` și `interpolator` - blocuri cu o rată fixă, unde numărul de date de intrare este un multiplu al numărului datelor de ieșire, respectiv numărul datelor de ieșire este un multiplu al celor de intrare. Pe lângă „semnătura” intrărilor și ieșirilor, care reprezintă structurarea lor pentru fiecare bloc, acestea primesc ca parametru și factorul de decimare, respectiv interpolare.
- `hierarchical` (bloc ierarhic) - este alcătuit din alte blocuri și, la instanțierea acestuia, el se ocupă și de instanțierea blocurilor componente.

Detaliile specifice implementării blocurilor în GNU Radio vor fi acoperite în Capitolul 3.

O metodă `work` este apelată pe baza elementelor de ieșire pe care este capabilă să le producă la un moment dat, iar `scheduler`-ul GNU Radio gestionează apelarea acestei metode pe baza cerințelor specifice ale unui bloc, precum numărul de date de intrare necesare producerii unei date de ieșire, și a stării `buffer`-elor de intrare sau de ieșire, prin trimiterea de comenzi și mesaje de stare între blocurile din lanțul de procesare și este cea mai complexă și remarcabilă parte a întregului sistem [21]. În plus, poate aplica restricții legate de alinierea datelor și ajustează pointerii datelor de intrare și de ieșire, în funcție de cum au fost „consumate”.

GNU Radio pune la dispoziție și o interfață grafică denumită **GNU Radio Companion** cu ajutorul căreia se pot vizualiza blocurile create, precum și cele deja existente în biblioteci, care se pot asambla într-un lanț de procesare. Cu ajutorul platformei Qt pentru dezvoltare a elementelor de interfață grafică, dezvoltatorul își poate crea propriile blocuri cu scopul de a vizualiza datele obținute. În plus, sunt deja disponibile o serie de astfel de blocuri, de exemplu pentru vizualizarea spectrului sau a semnalelor în domeniul timp, asemănător unui osciloscop digital.

În concluzie, GNU Radio reprezintă o opțiune atractivă pentru implementarea eficientă și simplă a blocurilor de procesare pentru SDR. El beneficiază de o documentație clară și de o comunitate deschisă, de foarte mare ajutor în rezolvarea problemelor care pot apărea în dezvoltare.

Capitolul 3

Implementarea software

3.1 Implementarea DoA folosită

Deoarece scopul proiectului de licență nu este implementarea integrală a întregului algoritm MUSIC, întrucât aceasta există deja în multiple forme, a trebuit să alegem o implementare deja existentă, integrată în GNU Radio, cu o structură clară, căreia să îi putem, apoi, evalua performanțele. Am ales o implementare creată de Ettus Research [4], care pun la dispoziție o aplicație ce dovedește capabilitățile de sincronizare ale dispozitivelor TwinRX. În Figura 3.1 este prezentat lanțul de procesare vizualizat cu ajutorul GNU Radio Companion. Ne vom referi la această implementare și sub numele `gr-doa`, care reprezintă și denumirea modulului, cu mențiunea că abrevierea *gr* provine de la denumirea GNU Radio, iar *DoA* (Direction of Arrival) este abrevierea în limba engleză pentru direcția de incidență.

3.2 Descrierea funcționalității

3.2.1 Pașii elementari ai algoritmului MUSIC

În Secțiunea 2.3 s-a făcut o detaliere a algoritmului MUSIC, împreună cu fundamentul matematic și demonstrațiile pe care se bazează. În continuare, este util să identificăm pașii elementari ai algoritmului, care se vor suprapune peste blocurile folosite în procesare. Reamintim faptul că M reprezintă numărul de elemente ale șirului de antene și N este numărul de semnale care ajung la fiecare dintre acestea.

Pasul 1

Știind că semnalul x_i ajunge la elementul cu numărul i al șirului de antene, matricea de autocorelație a intrării poate fi calculată astfel:

$$\mathbf{R}_{xx} = E[\mathbf{x}\mathbf{x}^H] \quad (3.1)$$

$$\mathbf{x} = [x_1 \quad x_2 \quad \dots \quad x_M] \quad (3.2)$$

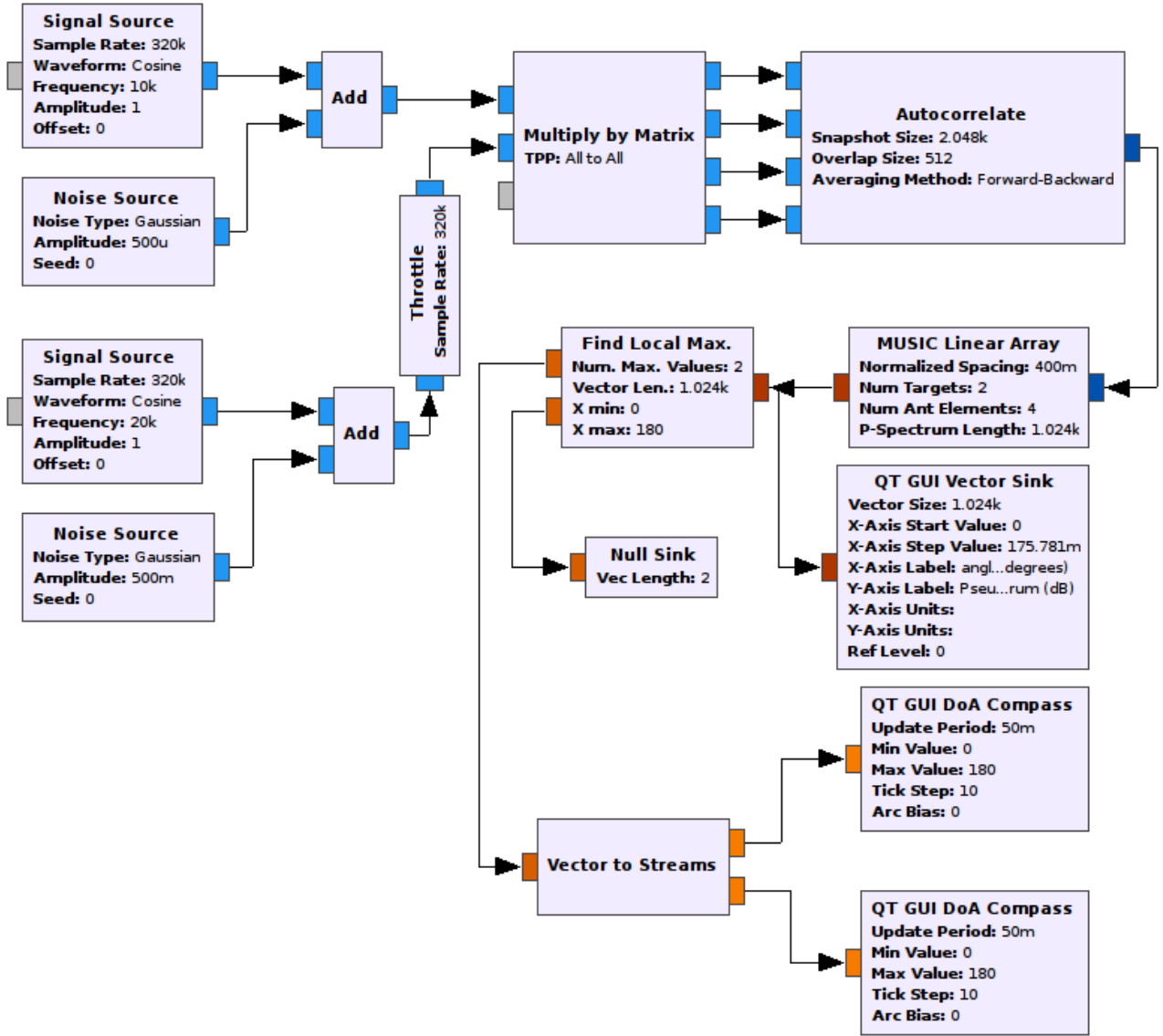


Figura 3.1: Lanțul de procesare din implementarea agloritmului MUSIC folosită

Pasul 2

Estimarea numărului de semnale care ajung la fiecare element al șirului de antene. Pentru a efectua acest pas, trebuie să calculăm valorile proprii ale matricei R_{xx} și, din ordinul de multiplicitate K al celei mai mici valori proprii, vom estima numărul de semnale astfel:

$$\hat{N} = M - K \quad (3.3)$$

Pasul 3

Calculăm spectrum spațial MUSIC folosindu-ne de Ecuația (3.4).

$$P_{MUSIC}(\theta) = \frac{\mathbf{a}^H(\theta)\mathbf{a}(\theta)}{\mathbf{a}^H(\theta)\mathbf{V}_N\mathbf{V}_N^H\mathbf{a}(\theta)} \quad (3.4)$$

$$\mathbf{V}_N = [v_{N+1}, \dots, v_M] \quad (3.5)$$

Pasul 4

Căutăm maximele spectrului MUSIC estimat, care ne dau direcțiile de incidență pentru cele \hat{N} semnale.

3.2.2 Descrierea blocurilor componente

Pentru a explica modul de lucru al lanțului de procesare, vom defini mai întâi o serie de parametri folosiți în blocurile componente. Numele acestora nu reflectă neapărat numele variabilelor folosite în cod, dar ne ajută să identificăm mai bine noțiunile de interes.

- `sample_rate` Rata la care este eșantionat semnalul
- `tone_freq_i` Frecvența semnalului de pe intrarea i
- `norm_spacing` Distanța normalată dintre elementele șirului de antene
- `num_targets` Numărul de semnale de intrare care ajung la șirul de antene, denumit în unele situații și N
- `num_array_elements` Numărul de elemente din șirul de antene, denumit în unele situații și M
- `spectrum_length` Numărul de elemente al spectrului MUSIC calculat
- `snapshot_size` Dimensiunea capturii, adică numărul de eșantioane folosite pentru calculul autocorelației, denumit și K
- `overlap_size` Numărul de eșantioane care se suprapun la calcularea unor valori succesive ale autocorelației

Cu cât sistemul dispune de un număr mai mare de antene, cu atât rezoluția spațială obținută va fi mai bună, iar cu cât dimensiunea capturii pentru autocorelație este mai mare, cu atât acuratețea detecției va crește, deoarece mai puține eșantioane în calculul autocorelației se traduc într-o corelație estimată mai puternică a semnalelor de intrare, lucru care nu este de dorit [22].

3.2.3 Datele de intrare

În configurația actuală, se folosesc două surse de semnal ($N = 2$) care generează forme de undă cosinusoidale cu frecvențe de 10 kHz și 20 kHz, peste care adăugăm zgomot Gaussian, și un șir de antene cu patru elemente ($M = 4$). Blocul **Throttle** este folosit pentru a limita volumul de date la frecvența semnalului de la intrare, așa cum s-ar comporta într-un caz real. Nefolosirea acestui bloc ar presupune consumarea tuturor resurselor de procesare disponibile pe dispozitivul de calcul. Particularizarea este făcută pentru a facilita explicarea blocurilor componente și nu îngreșează în niciun fel generalizarea ulterioară pentru un set diferit de parametri de intrare.

3.2.4 Blocul „Multiply by Matrix”

Acest bloc este folosit pentru a simula felul în care semnalele ajung la șirul de antene, înmulțind matricea colectoare a șirului cu un vector format din eșantioane ale semnalului de intrare. Dacă

\mathbf{A} este matricea dată ca parametru de intrare al blocului, de dimensiune $M \times N$, și \mathbf{X}_N este un vector coloană alcătuit din cele N intrări ale blocului, atunci rezultatul înmulțirii este:

$$\mathbf{Y}_M = \mathbf{A}\mathbf{X}_N, \quad (3.6)$$

unde \mathbf{Y}_M un vector coloană construit din ieșirile blocului. Prin urmare, blocul trebuie să aibă un număr de N intrări și M ieșiri.

Matricea colectoare are în vedere unghiurile de incidență și distanța normală dintre elementele șirului de antene. Distanța normală reprezintă distanța dintre antene exprimată în metri împărțită la lungimea de undă a purtătoarei. Conform [4], distanța normală trebuie să fie cel mult jumătate din lungimea de undă a semnalului, deoarece, în caz contrar, ar apărea fenomenul de aliere spectrală, care ar putea deteriora rezoluția algoritmului MUSIC.

În cazul nostru, matricea colectoare este

$$\mathbf{A} = [\mathbf{a}(\theta_1) \quad \mathbf{a}(\theta_2)]$$

unde $\mathbf{a}(\theta_i)$ este vectorul director corespunzător unghiului de incidență θ al semnalului i . Ieșirile blocului **Multiply by Matrix** reprezintă suma semnalelor care ajung sub diferite unghiuri de incidență la fiecare antenă.

3.2.5 Blocul "Autocorrelate"

Următorul bloc, **Autocorrelate**, corespunde pasului §1 al algoritmului MUSIC, deși calculează, de fapt, un estimat al autocorelației semnalului sub forma unei matrice de corelație eșantionată. O metodă de calcul a acestei matrice este de a colecta un număr de K eșantioane într-o perioadă de timp denumită „captură”, formând matricea \mathbf{X}_K de dimensiune $N \times K$, ceea ce conduce la următoarea formulă:

$$\mathbf{C}_x = \frac{1}{K} \mathbf{X}_K \mathbf{X}_K^H. \quad (3.7)$$

În [23] s-a sugerat faptul că adăugarea unui pas de mediere antegradă-retrogradă a matricei de corelație eșantionată va crește performanțele estimării unghiului de incidență, astfel încât calculul se ajustează după cum urmează:

$$\mathbf{C}_x = \frac{1}{2K} \mathbf{X}_K \mathbf{X}_K^H + \frac{1}{2K} \mathbf{J} \mathbf{X}_K^* \mathbf{X}_K^T \mathbf{J}, \quad (3.8)$$

unde \mathbf{J} este o matrice de reflexie (elementele diagonalei secundare sunt egale cu 1 și restul sunt egale cu 0).

Parametrii blocului **Autocorrelate** sunt:

- Dimensiunea de captură, denumită K , ce reprezintă numărul de eșantioane de intrare folosite în calculul matricei de corelație pentru un element de ieșire.
- Dimensiunea de suprapunere, adică numărul de eșantioane care se suprapun în calculul a două matrice de corelație succesive.
- Metoda de mediere, care poate fi antegradă-retrogradă, retrogradă, sau metoda standard antegradă.

3.2.6 Blocul „MUSIC Linear Array”

În această aplicație, presupunem cunoscut numărul de semnale de intrare, deci nu avem nevoie de un bloc separat pentru pasul §2. Prin urmare, trecem direct la blocul **MUSIC Linear Array** care calculează spectrul MUSIC din pasul §3.

În constructorul blocului se formează un vector colector care cuprinde toate unghiurile posibile dintr-o deschidere de 180 grade, cu o rezoluție de `1/spectrum_length`. Cu cât un unghi folosit în generarea vectorului colector este mai aproape de unghiul de incidență, cu atât mai aproape va fi spectrul MUSIC de 0. În teorie, când acestea coincid, spectrul MUSIC tinde la 0 când numărul de observații tinde la infinit. Prin urmare, pentru fiecare dată de intrare, un număr de `spectrum_length` valori ale spectrului MUSIC va trebui calculat, din care va fi păstrată doar partea reală, deoarece ne interesează doar amplitudinea spectrului MUSIC.

Pentru a calcula spectrul MUSIC, blocul primește ca intrare rezultatul autocorelației, care este descompus apoi în vectori proprii, și spectrul este calculat conform Formulei (2.16). Blocul primește ca parametri de intrare numărul de surse de semnal, numărul de antene, distanța normalată dintre ele și lungimea spectrului.

3.2.7 Blocul „Find Local Max”

Pentru ca aplicația să găsească valorile unghiurilor de incidență, este nevoie de blocul **Find Local Max**, care realizează pasul §4 al algoritmului. Blocul oferă la ieșire unghiurile la care sunt găsite maximele din spectru și caută exact N astfel de maxime. El oferă, de asemenea, informații despre amplitudinea maximelor, dar din moment ce ele nu sunt de interes pentru aplicație sunt direcționate către un bloc denumit **Null Sink**, care le înlătură.

3.2.8 Vizualizarea rezultatelor

Putem vizualiza spectrul fie imediat după blocul **MUSIC Linear Array**, precum în Figura 3.2, și să ne formăm o idee despre unghiul de incidență măsurat cu blocul **QT GUI Vector Sink** sau, folosind blocul **QT GUI DoA Compass**, putem vedea precis unghiul estimat pentru cele două semnale după procesarea din blocul **Find Local Max**. Se poate observa un exemplu de astfel de ieșire pentru unul dintre semnalele de intrare în Figura 3.3. Folosind două cursoare, putem schimba unghiurile de incidență în timp real și să observăm cum algoritmul se adaptează acestor schimbări.

3.3 Evaluarea profilului de execuție al implementării algoritmului MUSIC

Implementarea `gr-doa` folosește biblioteca Armadillo [24] pentru efectuarea operațiilor comune de algebră liniară în limbajul C++. Pentru descompunerea matricelor folosește biblioteca LAPACK [25] (Linear Algebra PACKage), care oferă rutine pentru rezolvarea sistemelor de ecuații liniare, probleme de valori proprii și descompunerea valorilor singulare ale unei matrice. LAPACK se folosește de BLAS (Basic Linear Algebra Subprograms) [26], o specificație care

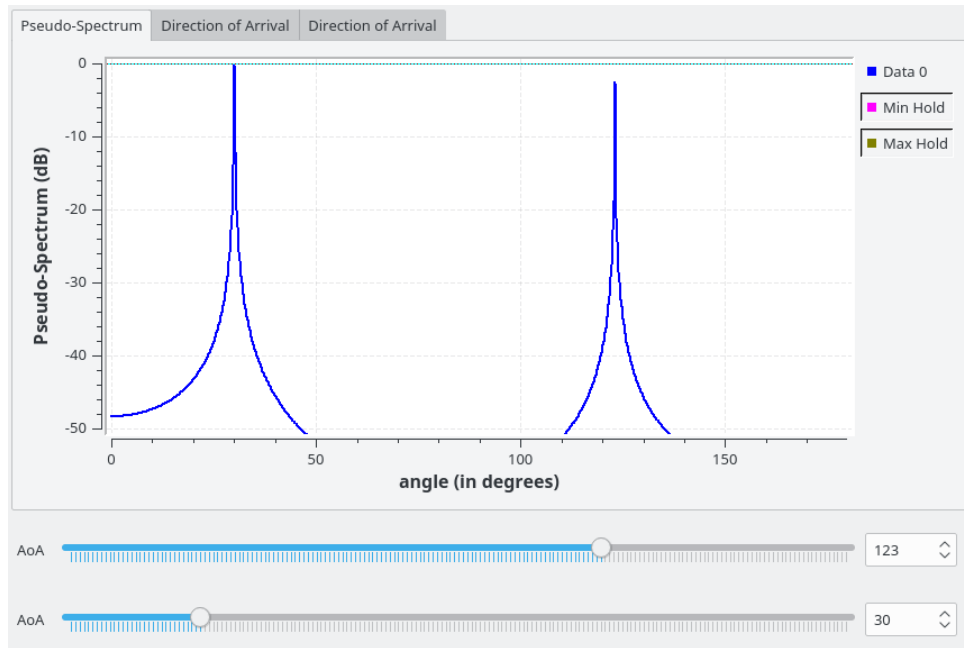


Figura 3.2: Spectrul MUSIC

descrie un set de rutine *low-level* pentru operații algebrice precum adunări de vectori, produs scalar, produs vectorial, combinații liniare și înmulțire de matrice. Există mai multe implementări posibile ale specificației BLAS, care se folosesc, în general, de instrucțiuni specifice sistemului folosit, precum suportul hardware pentru operații în virgulă mobilă sau instrucțiuni SIMD, ajungând la performanțe deosebite. Armadillo poate efectua înmulțiri de matrice și fără suportul BLAS, dar acestea au o performanță redusă și anumite descompuneri de matrice pot deveni indisponibile fără LAPACK și BLAS. În evaluarea performanțelor, ne-am asigurat că `gr-doa` folosește LAPACK și BLAS pentru atingerea unor performanțe maxime.

3.3.1 Metodologia de evaluare

Interfața grafică GNU Radio Companion generează un script Python bazat pe lanțul de procesare creat, care folosește apoi SWIG [27], un utilitar care îi permite să acceseze codul C++ și să execute graficul. Din cauza acestui pas intermediar, performanțele evaluate pe baza scriptului nu sunt foarte relevante, deoarece funcții folosite în procesul de interfațare a codului Python cu codul C++ ajung să consume o mare parte din timpul de execuție. Din acest motiv, soluția preferată a fost crearea unui program C++ care construiește lanțul de procesare într-un mod similar, ceea ce elimină și dependența de o interfață grafică.

În acest program am eliminat toate blocurile dependente de Qt și, unde a fost necesar, le-am înlocuit cu blocuri **Null Sink**, care pur și simplu ignoră datele primite, deci din acest punct de vedere nu avem operațiuni de I/O consumatoare de timp. În plus, blocul **Throttle** a fost eliminat, deoarece vrem să evaluăm programul atunci când este executat la viteză maximă, fără constrângeri.

Există două variante de execuție a lanțului de procesare, în funcție de tipul de surse de intrare folosite:

- Dacă se folosesc blocuri care generează anumite tipuri de semnal de intrare și de zgomot,

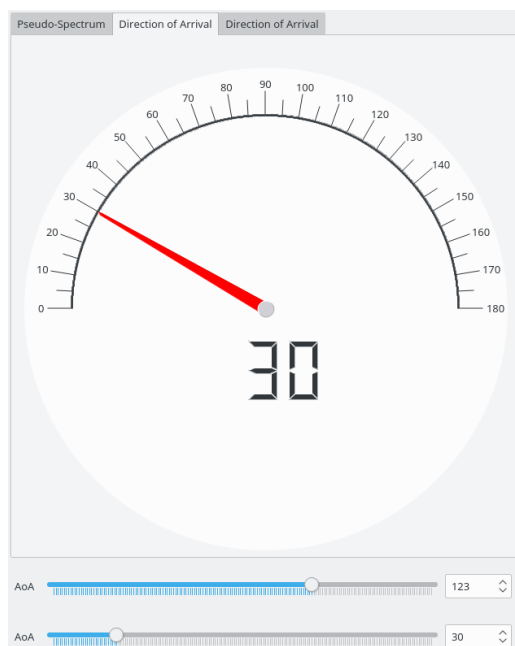


Figura 3.3: Unghiul de incidență estimat pentru unul dintre semnalele de intrare

se consideră că lanțul de procesare are un număr infinit de date de intrare și execuția sa se va termina dacă i se va încheia execuția în mod explicit după un anumit interval de timp. Dezavantajul, în acest caz, este faptul că generarea semnalelor consumă resurse de procesare importante, care pot introduce erori cu privire la profilul programului într-o situație reală în care am folosi antene fizice și semnalele de intrare nu ar trebui simulate. Din acest motiv, această variantă va fi folosită doar în testare.

- Dacă se folosesc blocuri care citesc semnalele de intrare din fișiere în care au fost salvate, în prealabil, eșantioane ale semnalelor dorite, se elimină procesarea nedorită din cazul anterior, deci vom folosi această variantă în generarea profilurilor de execuție. Execuția lanțului de procesare nu mai trebuie încheiată în mod explicit, deoarece acest lucru se va petrece automat, odată cu citirea tuturor datelor din fișierele sursă.

Pentru evaluarea profilului de execuție, am folosit **perf_events** [28], care realizează un *profil* al unui program în funcție de un eveniment dorit. Perf_events este un utilitar foarte puternic pentru măsurarea unui set mare de evenimente hardware, care permite examinarea codului de asamblare pentru funcțiile de interes, ceea ce poate fi de ajutor atunci când se urmăresc micro-optimizări. În acest caz, am fost interesați de procentul din numărul total de cicluri de execuție petrecut în diferite funcții, pentru a distinge punctele critice din execuția programului unde se pot aplica optimizări și care pot fi accelerate cu ajutorul procesorului ConnexArray.

Perf_events se bazează pe eșantionarea evenimentelor și o perioadă este exprimată în termeni de evenimente petrecute, nu de perioade de tact. El se folosește de numărătoare pe 64 biți emulate în software și colectează un eșantion atunci când numărătorul se resetează. Acest eșantion este un termen generic pentru un anumit tip de eveniment pentru care se realizează profilul. El reține informații despre punctul în care era programul în momentul colectării eșantionului, deci în momentul în care a fost întrerupt, sub forma unui *pointer la instrucțiune*.

În hardware, sistemele au implementate o unitate de monitorizare a performanței (Performance Monitoring Unit - PMU), care colectează statistici despre procesor și starea memoriei. Motivul

pentru care `perf_events` emulează un numărator pe 64 biți este faptul că PMU nu pune la dispoziție un astfel de registru hardware, iar acest lucru poate deveni important în momentul în care se interpretează locația unde a avut loc o întrerupere în cazul unui profil bazat pe eșantionarea întreruperilor, pentru că locul unde a avut loc o întrerupere PMU poate diferi de locul unde s-a resetat număratorul software.

În cazul de față, numărul de cicli petrecuți în funcțiile programului se pot obține folosind `perf record`, ce folosește în mod implicit ciclii ca tip de eveniment. Acest tip de eveniment generic este mapat la o implementare hardware specifică de către PMU care, pentru procesoarele Intel și sistemele AMD, nu menține o corelație constantă între eveniment și timp, din cauza unei scalări în frecvență (evenimentul se numește, în acest caz, *unhalted*), ceea ce înseamnă că ciclii nu sunt numărați atunci când procesorul nu lucrează (se află în starea *idle*). Prin urmare, această analiză ne oferă informații doar despre procesarea efectivă din anumite blocuri, nu și despre timpul pe care acestea îl petrec, de exemplu, așteptând date de intrare. Pentru un profil asupra acestui aspect se pot verifica evenimente precum *stalls* sau *cache misses*.

Recapitulând, folosind `perf_events`, putem obține informații despre ciclii petrecuți în funcțiile programului folosind comanda din Listarea 3.1, unde a fost specificat și tipul de eveniment ce se dorește înregistrat prin argumentul `-e cycles`. În plus, am adăugat și opțiunea `-call-graph dwarf`, care permite folosirea standardului de date pentru *debugging* DWARF pentru a obține informații despre *call stack* (stiva de apeluri).

```
1 perf record -e cycles --call-graph dwarf ./run_MUSIC_profile
```

Listarea 3.1: Comandă pentru profil realizat cu `perf`

Pentru a obține rezultate corecte, este important ca programul să fie compilat utilizând *frame pointers* (pointeri la cadre), care sunt, în general, omiși din motive de optimizare și fără de care este posibil să obținem un profil incomplet.

Lanțul de procesare este executat folosind mai multe fire de execuție, gestionate de GNU Radio, iar comanda de mai sus, folosită per proces, va agrega statisticile pentru fire de execuție la nivelul procesului, această opțiune de moștenire fiind activată în mod implicit.

3.3.2 Rezultatele evaluării pe un procesor Intel Core i7-6500U

Într-o primă etapă, performanțele implementării `gr-doa` au fost evaluate pe un procesor Intel Core i7-6500U, cu frecvența de 2.50 GHz pentru a identifica punctele cheie din execuție care pot fi optimizate.

Folosind nivelul minim de optimizare și activând simbolurile de *debugging*, cele mai importante rezultate (pentru funcțiile care consumă mai mult de 2% din numărul total de cicli) sunt prezentate în Tabelul 3.1.

Conform documentației [29], *Overhead* este procentul din numărul total de cicli petrecuți într-o anumită funcție, *Command* este numele *task*-ului care poate fi citit cu `/proc/<pid>/comm`, și *Symbol* este numele funcției aflate în execuție la momentul sondării.

Overhead	Command	Symbol
49,14%	find_local_max6	arma::glue_hist::apply_noalias<unsigned int>
13,73%	autocorrelate4	cgemm_
10,51%	MUSIC_lin_array	cgemm_
3,63%	multiply_matrix	__mulsc3
3,56%	MUSIC_lin_array	cgemv_
3,06%	MUSIC_lin_array	gr::doa::MUSIC_lin_array_impl::work
2,33%	MUSIC_lin_array	__logf_finite
2,08%	multiply_matrix	gr::blocks::multiply_matrix_cc_impl::work

Tabela 3.1: Profilul lanțului de procesare MUSIC realizat pe un procesor Intel Core i7

Investigând în continuare, am aflat că se petrece cel mai mult timp în funcția `arma::glue_hist::apply_noalias<unsigned int>`, din biblioteca **Armadillo**, care este apelată din funcția `find_local_max` și care se ocupă de găsirea maximelor din spectrul MUSIC.

Acest bloc poate fi implementat mai eficient, având în vedere că partea intensivă din punct de vedere computațional poate fi evitată în majoritatea cazurilor în care niciunul dintre punctele de maxim nu este egal $-\infty$.

Prin eliminarea acestei procesări acolo unde nu este necesară, obținem noile rezultate de performanță în Tabelul 3.2, unde au fost prezentate doar funcțiile care consumă mai mult de 5% din numărul total de cicli de execuție. O vizualizare mai intuitivă asupra acestor rezultate este oferită în Figura 3.4, realizată cu ajutorul programului FlameGraph [30]. Acesta construiește un grafic pe baza rezultatelor generate folosind `perf_events`, cu următoarea interpretare:

- Fiecare bloc reprezintă o funcție (*stack frame*).
- Pe axa verticală se află adâncimea stivei, adică ordinea de apelare a funcțiilor („descendenți”) din interiorul altor funcții (care reprezintă „părinți” acestora). Blocul care reprezintă funcția părinte se află întotdeauna sub blocul care reprezintă funcția descendent, deci funcția rădăcină se află pe nivelul cel mai de jos al graficului, iar funcția frunză pe cel mai înalt.
- Lățimea blocurilor este direct proporțională cu frecvența cu care au apărut în raporturile de activitate a stivei (*stack traces*).

Overhead	Command	Symbol
28,63%	autocorrelate9	cgemm_
23,16%	MUSIC_lin_array	cgemm_
5,40%	multiply_matrix	__mulsc3
5,04%	autocorrelate9	std::conj<float>

Tabela 3.2: Profilul lanțului de procesare MUSIC realizat pe un procesor Intel Core i7 după aplicarea corecțiilor în algoritmul de identificare a maximelor spectrului MUSIC

Observăm, în primul rând, că cele mai importante blocuri din punct de vedere al procesării sunt **Autocorrelate** și **MUSIC Linear Array**, detaliate în Secțiunile 3.2.5, și, respectiv, 3.2.6.

În ceea ce privește blocul **MUSIC Linear Array**, punctul în care se apelează rutina pentru înmulțirile de matrice se află în calculul spectrului MUSIC folosind Formula (2.16). Partea din cod care efectuează acest calcul se găsește la linia 136, în Listarea 3.3. În această porțiune de cod, matricea `d_vii_matrix` conține pe fiecare coloană vectorii directori, iar matricea `d_vii_matrix_trans` este transpusa și conjugata acesteia. Matricea `U_N_sq` este obținută din produsul $\mathbf{V}_N \mathbf{V}_N^H$, unde \mathbf{V}_N este format din vectorii proprii ai matricei de covarianță care sunt perpendiculari pe subspațiul semnalelor.

Este important, în acest caz, faptul că matricea `d_vii_matrix` este calculată o singură dată, în constructorul blocului, iar fiecare coloană din acest vector este înmulțită cu aceeași matrice calculată pe baza intrării blocului. Avem, așadar, un număr egal cu dimensiunea spectrului MUSIC de înmulțiri în lanțuite dintre un vector linie, o matrice și un vector coloană a căror dimensiune depinde de numărul de antene din sistem, deci numărul total de înmulțiri va fi relativ mare în comparație cu dimensiunea tablourilor care se înmulțesc.

```

134 for (int ii = 0; ii < d_pspectrum_len; ii++)
135 {
136     Q_temp =
137         as_scalar(d_vii_matrix_trans.row(ii) * U_N_sq * d_vii_matrix.col(ii));
138     out_vec(ii) = 1.0 / Q_temp.real();
139 }

```

Listarea 3.3: Punctul critic din blocul MUSIC Linear Array

3.3.3 Rezultatele evaluării pe placa de dezvoltare Xilinx ZedBoard Zynq-7000 ARM/FPGA SoC

Am evaluat performanțele și pe placa de dezvoltare Zedboard Zynq-7000, prezentată în Secțiunea 2.4, și am ajuns la concluzia că procentajele obținute sunt asemănătoare.

Tabela 3.3 prezintă rezultatele care consumă mai mult de 5% din timpul total de procesare al evaluării pe procesorul Dual-Core ARM Cortex A9, realizată în aceeași configurație ca și cea de pe procesorul Intel.

Overhead	Command	Symbol
18,97%	autocorrelate9	cgemm_
13,68%	MUSIC_lin_array	cgemm_
5,93%	autocorrelate9	gr::doa::autocorrelate_impl::general_work
5,79%	MUSIC_lin_array	cgemv_
5,76%	MUSIC_lin_array	gr::doa::MUSIC_lin_array_impl::work

Tabela 3.3: Profilul lanțului de procesare MUSIC realizat pe un procesor ARM Cortex A9 după aplicarea corecțiilor în algoritmul de identificare a maximelor spectrului MUSIC

3.4 Testare

Testarea întregului lanț de procesare este o etapă esențială nu numai pentru determinarea corectitudinii rezultatelor finale, dar și pentru evaluarea preciziei obținute. Conceptul de *unit testing* este o metodă utilă în acest scop, cu care se pot testa blocuri individuale sau agregate ale unui produs software, fiind un instrument indispensabil în industria software.

Deși GNU Radio are încorporate metode pentru testarea blocurilor dezvoltate, nu oferea suficientă flexibilitate pentru a putea fi utilizate în testarea folosind simulatorul sau suportul hardware a kernelurilor de accelerare într-o etapă viitoare, motiv pentru care a trebuit să recurgem la altă soluție. Biblioteca **Google Test** [31] s-a dovedit a fi o soluție mai bună în acest sens, testele putând fi scrise cu ușurință și executate cu ajutorul unui script care pornește simulatorul, atunci când este cazul, și prin intermediul căruia pot fi oferite căile pentru cozile de comunicare cu acceleratorul ca argumente în linia de comandă către program.

În Google Test se definesc *asertiuni*, adică afirmații a căror valoare de adevăr trebuie stabilită. Aceste asertiuni sunt folosite de teste pentru a verifica comportamentul codului: dacă rezultatele obținute sunt diferite de cele așteptate sau dacă se întâlnește altă eroare în timpul execuției codului, un test eșuează, iar în caz contrar se consideră trecut. Testele pot fi grupate în *test case*-uri, iar un program de test poate să cuprindă mai multe astfel de *test case*-uri.

Lanțul de procesare are nevoie de o serie de parametri prezentați în Secțiunea 3.2.2, la care se vor adăuga cozile prin care se va comunica cu procesorul ConnexArray și, eventual, alți parametri cu care să putem controla forma de undă a semnalului de intrare sau tipul de zgomot care se adaugă peste acesta. Definim o clasă `flowgraph_parameters` care are ca date membre acești parametri de intrare și metode pentru setarea valorilor acestora. O altă clasă, `doa_flowgraph`, are un membru de tip `flowgraph_parameters` și o metodă prin care construiește un *flowgraph* cu parametrii primiți prin intermediul acesteia. În Anexa 1.1 se găsește codul sursă al fișierului header care conține declarațiile claselor menționate.

Metoda care construiește lanțul de procesare folosit în testare este similară cu cea descrisă în Secțiunea 3.3.1 pentru evaluarea performanțelor, cu excepția faptului că datele de ieșire nu mai sunt ignorate, ci direcționate către un bloc care le va returna la ieșire, pentru a fi preluate de către testele definite și apoi verificate. Codul care implementează construcția *flowgraph*-ului pentru două semnale de intrare se găsește în Anexa 1.2.

În Listarea 3.4 este prezentat un exemplu de definire a unui test, în care mai întâi se construiește o instanță a clasei de parametri cu datele dorite, care este apoi folosită pentru a crea clasa responsabilă de lanțul de procesare. Se folosește tipul de asertiune `EXPECT_THAT`, care compară valorile unghiurilor de incidență găsite cu cele specificate în parametrii de intrare, cu o anumită eroare a cărei valoare poate fi controlată. De exemplu, dacă sursele au unghiuri de incidență foarte apropiate, dacă unghiul de incidență se află la limitele domeniului, sau dacă raportul semnal-zgomot este foarte mic, ne putem aștepta la o precizie mai redusă.

Am definit mai multe clase de teste care verifică acuratețea rezultatelor în următoarele situații:

- Sursele provin din unghiuri relativ distante, caz în care precizia ar trebui să fie foarte

bună, cu o eroare de mai puțin de $\varepsilon = 0.2^\circ$.

- Direcțiile de incidență ale celor două surse sunt foarte apropiate (între 10° și 2°) sau se află la limitele intervalului de $[0^\circ, 180^\circ]$, caz în care precizia scade cu până la $\varepsilon = 1^\circ$ în cazurile extreme.
- Frecvențele semnalelor sunt foarte apropiate, dar în acest caz nu s-au observat modificări de precizie semnificative, cel puțin în simulări.
- Diferite dimensiuni ale spectrului MUSIC, care modifică rezoluția obținută, deci afectează și precizia.
- Diferite dimensiuni ale capturii pe care se calculează autocorelația, cu același efect asupra preciziei.
- Diferite configurații de antene.

```

1 TEST(DoaTwoSourcesTest, GeneralTest) {
2     auto fg_params = flowgraph_parameters<2>()
3         .set_sample_rate(320000)
4         .set_norm_spacing(0.4)
5         .set_num_array_elements(4)
6         .set_p_spectrum_length(1024)
7         .set_snapshot_size(2048)
8         .set_overlap_size(512)
9         .set_nr_output_items(1024)
10        .set_freq({10000, 20000})
11        .set_theta_deg({40, 120})
12        .set_signal_amplitude({1, 1})
13        .set_noise_amplitude({0.005, 0.00005})
14        .set_waveform({gr::analog::GR_COS_WAVE,
15                      gr::analog::GR_COS_WAVE})
16        .set_noise_type({gr::analog::GR_GAUSSIAN,
17                        gr::analog::GR_GAUSSIAN})
18        .set_distributionFIFO(my_argv[1])
19        .set_reductionFIFO(my_argv[2])
20        .set_writeFIFO(my_argv[3])
21        .set_readFIFO(my_argv[4]);
22    auto doa_fg = doa_flowgraph<2>(fg_params);
23    EXPECT_THAT(doa_fg.build_flowgraph(), UnorderedElementsAre(
24        FN_HIGH_PRECISION(40),
25        FN_HIGH_PRECISION(120)));
26 }

```

Listarea 3.4: Exemplu de test realizat cu Google Test

3.5 Concluzii

Având în vedere rezultatele obținute, putem concluziona că punctele critice în procesarea algoritmului MUSIC au loc la înmulțirile de tablouri unidimensionale sau bidimensionale de numere complexe, fie că este vorba de înmulțirea dintre un vector și conjugatul său, ca în cazul autocorelației, fie că vorbim de înmulțirea dintre un vector și o matrice, sau de un produs înălțuit dintre un vector, o matrice și un alt vector. Așadar, merită să ne îndreptăm atenția spre implementarea unor kerneluri pentru procesorul ConnexArray care să efectueze aceste operații, care vor fi detaliate în Capitolul 4, urmând a le evalua performanțele în Capitolul 5.

Așa cum a fost subliniat la începutul Secțiunii 3.3, înmulțirile dintre tablouri de date sunt efectuate utilizând implementări ale specificației BLAS, cu suport în implementarea hardware a sistemului pe care se lucrează. Ele exploatează ierarhia memoriei unui sistem, localizarea

datelor (*data locality*), împărțind datele pe care se operează în blocuri care sunt transferate în memoria de pe diferite niveluri. Cu cât un tip de memorie este mai rapidă, cu atât ea dispune de un spațiu de stocare mai mic; o clasificare a memoriilor în funcție de viteză, începând de la cea mai rapidă, este următoarea: registre, cache (care poate fi, la rândul său, împărțit pe mai multe niveluri), memoria principală, memoria secundară (*disk*). Dacă tablourile de date pe care se operează au dimensiuni foarte mari, ele nu vor putea fi aduse în întregime în cea mai rapidă memorie și de aceea este necesar să se opereze pe blocuri de date care să circule între tipurile de memorii enumerate, caz în care transferul I/O va consuma mai mult timp de execuție.

Având în vedere acest raționament, putem presupune că în implementarea kernelurilor vom avea un avantaj suplimentar atunci când dimensiunile matricelor vor fi mai mari, comparativ cu implementarea BLAS. Chiar și când acesta nu va fi cazul, vom încerca să efectuăm mai multe înmulțiri simultan, pentru un grad mai mare de paralelizare.

Un alt aspect interesant este și faptul că în anumite situații se poate exploata și dimensiunea tablourilor de date, astfel că, de exemplu, biblioteca OpenBLAS [32] conține instrucțiuni specifice pentru diferite tipuri de procesoare care iau în calcul și dimensiunea matricelor pentru un plus de performanță [33]. Prin urmare, acolo unde este cazul, vom lua în considerare adaptarea algoritmilor la anumite dimensiuni ale unor matrice pentru a le crește performanța.

Referitor la întregul ansamblu de procesare care implementează algoritmul MUSIC, trebuie menționat faptul că este posibil ca optimizarea unui singur bloc poate fi „mascată” în evaluarea performanțelor de existența altui bloc mai lent, care va încetini întregul lanț de procesare. Cu alte cuvinte, blocurile sunt dependente unul de celălalt prin schimbul de date care are loc între ele și, în cazul în care un bloc reprezintă un *bottleneck*, performanțele întregului lanț de procesare vor avea de suferit din această cauză.

Capitolul 4

Accelerarea hardware

4.1 Aspecte specifice în implementarea kernelurilor

Procesorul ConnexArray permite lucrul atât cu tipuri de date fracționare în virgulă fixă, cât și în virgulă mobilă, cu diferențe în ceea ce privește precizia rezultatelor și timpul de calcul.

Datele în virgulă fixă au avantajul că se procesează rapid, operațiile de adunare, scădere, înmulțire sau depalare efectuate pe ele durând doar un ciclu de ceas, la fel ca în cazul numerelor întregi. În realitate, procesorul nu diferențiază numerele întregi de numerele fracționare în virgulă fixă, lăsând în sarcina programatorului interpretarea acestora în funcție de context. Pentru conversia din date în virgulă mobilă, cum sunt cele utilizate în general în limbajul C++, la cele în virgulă fixă necesare acceleratorului ConnexArray, este necesară o pre-scalare cu un anumit factor, ales în funcție de gama dinamică a numerelor, și o conversie la tipul de date întreg fără semn pe 16 biți (`uint16_t`).

În ceea ce privește datele în virgulă mobilă, nu există suport hardware pentru acestea, iar calculul se bazează pe o emulare software, motiv pentru care operațiile în virgulă mobilă pe procesor sunt foarte lente (de exemplu, o înmulțire între două date în virgulă mobilă poate consuma până la 50 de cicli de ceas).

Prin urmare, având în vedere că performanța programului este punctul principal de interes și că datele de intrare au o gamă dinamică destul de mică, în intervalul $(-2, 2)$, am ales reprezentarea datelor în virgulă fixă.

Funcționarea cozilor pentru comunicarea cu acceleratorul a fost explicată în Secțiunea 2.4, dar este important de precizat un aspect referitor la tipurile de date cu care lucrează. Un rezultat urmat în urma unei procesări pe accelerator poate fi citit fie din memoria sa locală, prin metoda `writeDataToArray`, caz în care citirea se face prin intermediul cozii `readFIFO`, fie, dacă acest rezultat este urmarea unei reducții, poate fi citit direct din coada `reductionFIFO` cu ajutorul metodei `readReduction`. În primul caz, rezultatele vor fi pe 16 biți, fără semn, urmând ca interpretarea lor cu semn să se facă în funcție de factorul cu care au fost scalate datele, iar în al doilea caz rezultatul va fi pe 32 biți, cu semn.

În esență, există două considerente principale care trebuie luate în calcul în conceperea unui

kernel:

- **Gestionarea datelor**, care presupune aranjarea datelor în memoria locală a acceleratorului, eventuala lor împărțire în blocuri de date care vor fi procesate în aceeași execuție a kernelului, când se face conversia lor din virgulă mobilă în virgulă fixă (și invers) și unde, momentul scrierii și citirii lor.
- **Algoritmul propriu-zis** implementat de kernel, care asumă deja o anumită aranjare anterioară a datelor.

Deși primul considerent este legat mai mult de programul principal și nu de mediul de lucru OPINCAA, el este puternic corelat cu procesarea pe accelerator, deci nu poate fi tratat separat și poate avea un impact major asupra performanței toate a procesării.

Un alt factor care trebuie luat în considerare este cel al dimensiunii tablourilor cu care se operează. În cazul ideal, performanțele maxime se obțin dacă acestea au ca dimensiuni puteri ale lui 2, care va facilita aranjarea lor pe liniile de procesare. Memoria locală, la modul general, are o capacitate de M linii a câte N elemente, unde N este numărul de elemente de procesare, structură ilustrată în Figura 4.1. Se observă cum un anumit PE_i are acces la elementele $LS[k][i]$ din memoria locală, unde $i = \overline{0, N-1}$, $k = \overline{0, M-1}$. În cazul de față, acceleratorul are o configurație cu $N = 128$ elemente de procesare și o memorie locală cu $M = 1024$ linii.

	PE_0	PE_1	PE_2	...	PE_{N-2}	PE_{N-1}
$LS[0]$						
$LS[1]$						
...						
$LS[M-1]$						
$LS[M-2]$						

Figura 4.1: Structura memoriei locale a acceleratorului ConnexArray

Observăm că cele mai importante limitări în procesarea datelor pe accelerator constă în pregătirea datelor de intrare și de ieșire și capacitatea memoriei locale, care constrânge numărul de date care pot fi procesate într-o execuție a unui kernel. Prin urmare, dacă avem un număr de date de intrare de procesat mai mare decât capacitatea memoriei locale a acceleratorului, acestea trebuie împărțite în blocuri de procesare de dimensiune mai mică (și, preferabil, de dimensiune egală). Avantajul procesării în blocuri cât mai mari de date este faptul că operațiile de I/O vor fi mai puține și se va elimina, astfel, un *overhead* important.

În plus, un avantaj este dat de faptul că putem pregăti datele de intrare pentru procesarea blocului următor (sau realiza alte operații) în timpul în care, în mod normal, am aștepta datele de ieșire de la accelerator. Acest lucru se poate face fie în același fir de execuție, întrucât lansarea în execuție a unui kernel nu este o operație blocantă, spre deosebire de citirea datelor, sau lansarea unui alt fir de execuție care să realizeze procesarea dorită și, dacă este necesar, sincronizarea lui cu firul principal de execuție.

În continuare, vom prezenta câteva soluții de realizare a kernelurilor pentru problemele identificate în modulul `gr-doa` care au la bază înmulțiri de tablouri de date. Astfel, Secțiunea 4.2 prezintă înmulțirea vectorilor de numere complexe, Secțiunea 4.3 dezvoltă înmulțirea unui vector cu o matrice pătratică, în Secțiunea 4.4 a fost elaborat un algoritm pentru o înmulțire înlănțuită dintre un vector, o matrice pătratică și un vector, iar în Secțiunea 4.5 este prezentată o modalitate de realizare a autocorelației unui semnal de intrare. În final, am descris modalitatea de integrare a acestor kerneluri în mediul de dezvoltare GNU Radio în Secțiunea 4.6.

4.2 Înmulțirea vectorilor de numere complexe pe procesorul ConnexArray

Considerăm, mai întâi, un caz general în care înmulțim un vector linie X cu un vector coloană Y , de aceeași dimensiune N pe care, pentru simplitate, o presupunem egală jumătate din numărul de elemente de procesare ale acceleratorului.

$$\mathbf{X} \triangleq [x_0 \ x_1 \ \dots \ x_{N-1}] \quad (4.1)$$

$$\mathbf{Y} \triangleq [y_0 \ y_1 \ \dots \ y_{N-1}]^T \quad (4.2)$$

Vectorii conțin numere complexe, deci elementele lor vor fi de forma:

$$x_i = a_i + jb_i, \quad i = \overline{0, N-1} \quad (4.3)$$

$$y_i = a'_i + jb'_i, \quad i = \overline{0, N-1} \quad (4.4)$$

Rezultatul înmulțirii celor doi vectori va fi o matrice cu un singur element complex.

$$\mathbf{R} \triangleq \mathbf{XY} = \left[\sum_{i=0}^{N-1} x_i y_i \right] = \left[\sum_{i=0}^{N-1} (a_i a'_i - b_i b'_i) + j \sum_{i=0}^{N-1} (a_i b'_i - a'_i b_i) \right] \quad (4.5)$$

Propunem, pentru calculul rezultatului, metoda de aranjare a elementelor din Figura 4.2, care figurează și următorii pași de procesare:

1. Elementele din primul vector sunt încărcate în locații succesive din memoria locală a acceleratorului, părțile lor reale alternând cu cele imaginare, iar apoi sunt transferate într-un registru (în cazul de față R1), procedând în mod asemănător și pentru cel de-al doilea (pentru acesta, vor fi transferate în registrul R2). În acest mod, părțile reale ale elementelor celor doi vectori se vor afla în PE cu număr de ordine par, iar cele imaginare în PE cu număr de ordine impar.
2. Efectuăm înmulțirea dintre conținutul registrelor R1 și R2, obținând astfel produsele necesare pentru calculul părții reale a rezultatului, care vor fi păstrate în registrul R3.
3. Se inversează semnul produselor de părți imaginare din registrul R3 doar pe PE-urile impare.
4. Se aplică o reducere pe registrul R3, obținând partea reală a rezultatului final.

5. Pentru a calcula partea imaginară a rezultatului, va trebui să aducem părțile imaginare ale elementelor din cel de-al doilea vector în aceleași PE-uri cu părțile reale ale primului, păstrând rezultatul deplasării în registrul R4.
6. Reciproc, părțile reale ale elementelor celui de-al doilea vector sunt aduse în aceleași PE-uri cu părțile imaginare ale elementelor primului vector și stocăm rezultatul deplasării în registrul R5.
7. Efectuăm o înmulțire între registrele R1 și R4 și suprascriem rezultatul în registrul R4.
8. Efectuăm o înmulțire între registrele R2 și R5 și suprascriem rezultatul în registrul R5.
9. În acest moment, avem produsele necesare pentru calculul părții imaginare a rezultatului final. Observăm, din Figura 4.2, că vom obține rezultate de interes doar în PE-urile cu index par pentru registrul R4 și în cele cu index impar pentru cele din registrul R5, deci, pentru un rezultat corect, putem muta doar conținutul registrului R5 din PE-urile impare în registrul R4.
10. Toate produsele parțiale pentru partea imaginară a rezultatului se află acum în registrul R4 și vom finaliza calculul acestuia prin aplicarea unei reducții pe acesta.

		PE ₀	PE ₁	PE ₂	PE ₃	PE ₄	PE ₅	PE _{2N-2}	PE _{2N-1}
Pasul 1	R1	x ₀	y ₀	x ₁	y ₁	x ₂	y ₂	x _{N-1}	y _{N-1}
	R2	x ₀ '	y ₀ '	x ₁ '	y ₁ '	x ₂ '	y ₂ '	x _{N-1} '	y _{N-1} '
Pasul 2	R3	x ₀ x ₀ '	y ₀ y ₀ '	x ₁ x ₁ '	y ₁ y ₁ '	x ₂ x ₂ '	y ₂ y ₂ '	x _{N-1} x _{N-1} '	y _{N-1} y _{N-1} '
Pasul 3	R3	x ₀ x ₀ '	-y ₀ y ₀ '	x ₁ x ₁ '	-y ₁ y ₁ '	x ₂ x ₂ '	-y ₂ y ₂ '	x _{N-1} x _{N-1} '	-y _{N-1} y _{N-1} '
Pasul 5	R4	y ₀ '	x ₁ '	y ₁ '	x ₂ '	y ₂ '	x ₃ '	y _{N-1} '	
Pasul 6	R5		x ₀ '	y ₀ '	x ₁ '	y ₁ '	x ₂ '	y _{N-2} '	x _{N-1} '
Pasul 7	R4	x ₀ y ₀ '		x ₁ y ₁ '		x ₂ y ₂ '		x _{N-1} y _{N-1} '	
Pasul 8	R5		y ₀ x ₀ '		y ₁ x ₁ '		y ₂ x ₂ '		y _{N-1} x _{N-1} '
Pasul 9	R4	x ₀ y ₀ '	y ₀ x ₀ '	x ₁ y ₁ '	y ₁ x ₁ '	x ₂ y ₂ '	y ₂ x ₂ '	x _{N-1} y _{N-1} '	y _{N-1} x _{N-1} '

REDUCE

REDUCE

Figura 4.2: Înmulțirea a doi vectori de numere complexe pe ConnexArray

Kernelul care implementează algoritmul descris se află în Anexa 2.1 Pentru eficiență, pașii nu se execută în ordinea explicată mai sus ci, de exemplu, se pot combina pașii 3 și 9 pentru reducerea timpului de calcul. O îmbunătățire adițională poate consta în rearanjarea instrucțiunilor astfel încât să se evite efectuarea unui `NOP` acolo unde este posibil, dar acest lucru reduce lizibilitatea codului, motiv pentru care va fi realizată doar în implementare.

În cazurile destul de comune întâlnite în practică în care dimensiunea unui vector (N) va fi mai mică decât numărul de PEs disponibile, putem, să efectuăm mai multe înmulțiri în aceeași execuție a kernelului. Modificarea va consta în faptul că reducția va trebui, în acest caz, efectuată în blocuri de dimensiune $2 \cdot N$.

În cazul contrar, în care N depășește capacitatea unei linii a acceleratorului, algoritmul propus poate funcționa în continuare dacă la „pregătirea” datelor de ieșire luăm în considerare faptul că mai multe rezultate de reducere trebuie însumate pentru formarea unui element al matricei de ieșire. Dacă N nu este un multiplu al capacității liniilor de procesare, elementele rămase neocupate dintr-o linie să fie inițializate cu zero pentru a nu afecta rezultatul reducerii, procedeu cunoscut sub numele de *zero padding*.

4.3 Kernel pentru înmulțirea unui vector cu o matrice

Unul dintre punctele critice ale implementării algoritmului MUSIC a fost găsit în blocul **MUSIC Linear Array**, detaliat în Secțiunea 3.2.6, la calculul spectrului MUSIC utilizând formula de mai jos.

$$P_{MUSIC}(\theta) = \frac{1}{\mathbf{a}^H(\theta) \mathbf{V}_N \mathbf{V}_N^H \mathbf{a}(\theta)} \quad (4.6)$$

Avem de a face, prin urmare, cu o înmulțire înlanțuită dintre un vector linie $\mathbf{a}^H(\theta)$ de lungime M , o matrice pătratică formată din produsul $\mathbf{V}_N \mathbf{V}_N^H$ pe care o vom nota \mathbf{V}_{Nsq} , de dimensiune $M \times M$ și un vector coloană $\mathbf{a}(\theta)$ de lungime M . Vom nota rezultatul primei părți a înmulțirii $X_M \triangleq \mathbf{a}^H(\theta) \mathbf{V}_{Nsq}$.

Am decis să calculăm X_M pe procesorul SIMD și partea finală $X_M \mathbf{a}(\theta)$ va fi lăsată ca discuție din considerente de performanță, urmând a stabili dacă este mai eficient să fie calculată pe procesorul ARM sau dacă este de preferat varianta efectuării întregului produs de trei matrice pe procesorul ConnexArray, care va fi descrisă în Secțiunea 4.4.

Mai întâi, considerăm cazul general al înmulțirii dintre un vector linie de lungime M cu o matrice pătratică de dimensiune $M \times M$, din care rezultă un vector linie de lungime M .

$$X_M \triangleq [\bar{a}_0 \quad \bar{a}_1 \quad \dots \quad \bar{a}_{M-1}] \begin{bmatrix} v_{0,0} & v_{0,1} & \dots & v_{0,M-1} \\ v_{1,0} & v_{1,1} & \dots & v_{1,M-1} \\ \vdots & \vdots & \ddots & \vdots \\ v_{M-1,0} & v_{M-1,1} & \dots & v_{M-1,M-1} \end{bmatrix} \quad (4.7)$$

$$X_M = \left[\sum_{i=0}^{M-1} \bar{a}_i v_{i,0} \quad \sum_{i=0}^{M-1} \bar{a}_i v_{i,1} \quad \dots \quad \sum_{i=0}^{M-1} \bar{a}_i v_{i,M-1} \right], \text{ unde} \quad (4.8)$$

$$a_i = x_i + jy_i, \quad i = \overline{0, M-1} \quad (4.9)$$

$$\bar{a}_i = x_i - jy_i \stackrel{\text{not}}{=} x_i + j\bar{y}_i, \quad i = \overline{0, M-1} \quad (4.10)$$

$$v_{i,k} = x_{i,k} + jy_{i,k}, \quad i = \overline{0, M-1}, k = \overline{0, M-1} \quad (4.11)$$

Rezultatul poate fi scris în continuare:

$$X_M = \begin{bmatrix} \sum_{i=0}^{M-1} (x_i x_{i,0} - \bar{y}_i y_{i,0}) + j \sum_{i=0}^{M-1} (x_i y_{i,0} + \bar{y}_i x_{i,0}) \\ \sum_{i=0}^{M-1} (x_i x_{i,1} - \bar{y}_i y_{i,1}) + j \sum_{i=0}^{M-1} (x_i y_{i,1} + \bar{y}_i x_{i,1}) \\ \dots \\ \sum_{i=0}^{M-1} (x_i x_{i,M-1} - \bar{y}_i y_{i,M-1}) + j \sum_{i=0}^{M-1} (x_i y_{i,M-1} + \bar{y}_i x_{i,M-1}) \end{bmatrix}^T \quad (4.12)$$

În mod evident, implementarea acestui produs se poate face extinzând algoritmul de înmulțire a doi vectori, unul linie și celălalt coloană, de aceeași dimensiune, deoarece înmulțirea unui vector cu o matrice constă, de fapt, în înmulțirea vectorului de intrare cu alți vectori formați din coloanele matricei de intrare, după cum se poate observa în Figura 4.3.

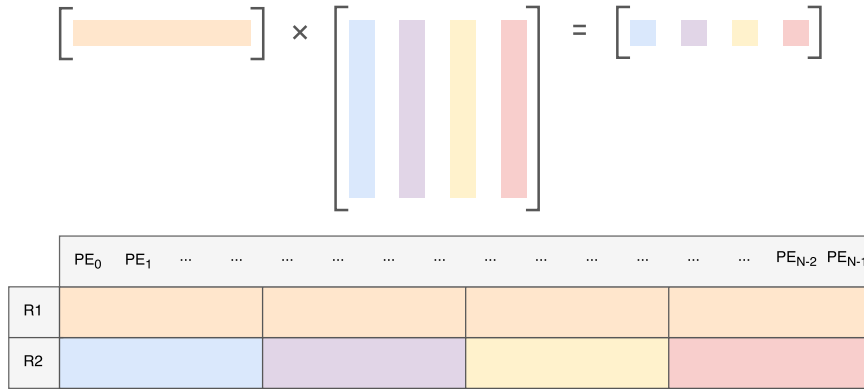


Figura 4.3: Exemplu de aranjare a elementelor în ConnexArray pentru o înmulțire dintre un vector și o matrice

Putem să distingem mai multe variații ale metodei propuse în funcție de numărul de elemente al matricei. Dacă notăm `arr_size_c` = $M \times 2$ și `mat_size_c` = $M \times M \times 2$ numărul de elemente în care am separat părțile reale și imaginare ale matricei, respectiv vectorului de intrare și `vector_array_size` numărul de elemente de procesare ale acceleratorului, avem următoarele posibilități de aranjare a datelor de intrare în memoria locală a acceleratorului:

- a. `mat_size_c` ≤ `vector_array_size` - în acest caz, cel puțin o matrice poate fi încărcată într-o linie din LS, coloană cu coloană. Deoarece vectorul de intrare se înmulțește cu fiecare coloană, el va fi copiat de M ori și stocat într-o altă linie din LS. Numărul de înmulțiri dintre vectori linie și matrice care pot fi efectuate simultan pe accelerator este egal cu numărul de matrice care încap într-o linie de procesare. În Figura 4.4 am prezentat aranjarea elementelor în ConnexArray în cazul în care se efectuează simultan două înmulțiri dintre doi vectori de intrare și aceeași matrice, unde $V.col(i)$ reprezintă un vector format din coloana i a matricei de intrare.

În general, vom dori să efectuăm înmulțiri pe blocuri mai mari de date într-o execuție a unui kernel, în care vom ocupa mai multe linii din LS, care vor fi apoi aduse, pe rând, în registre și prelucrate în mai multe iterații. Un exemplu de aranjare a vectorilor în această situație este prezentat în Figura 4.5a, în care s-a luat în considerare faptul că elementele nu vor ocupa toată capacitatea acceleratorului și trebuie adăugate elemente de gardă

pentru aliniere (*padding*). Este de ajuns ca matricea să fie încărcată într-o singură linie de procesare, restul fiind ocupate cu vectorii cu care aceasta se înmulțește.

În practică, pentru un accelerator o capacitate de 128 de elemente de procesare, se pot prelucra matrice cu dimensiunea maximă 8×8 care, în contextul algoritmului MUSIC, corespund unei configurații cu cel mult opt antene de recepție.

Vor fi efectuați următorii pași în kernel, corespunzători pseudocodului din Listarea 4.1:

1. Încărcăm linia cu matricea de intrare într-un registru.
2. La începutul fiecărei iterații pentru fiecare LS din blocul de procesare, încărcăm o linie din LS cu vectori de intrare într-un registru. În interiorul fiecărei iterații:
 - 2.1. Efectuăm înmulțirea dintre registrele din fiecare PE descrisă în Secțiunea 4.2.
 - 2.2. Realizăm reducții în blocuri de câte `arr_size_c` elemente pentru fiecare coloană a fiecărei matrice din linia de procesare.
 - 2.3. Incrementăm registrul folosit în adresarea liniilor din LS ce conțin vectori și trecem la următoarea iterație pentru fiecare LS din blocul de procesare.

Codul kernelului care implementează algoritmul descris se află în Anexa 2.2.

- b. `mat_size_c > vector_array_size` și `arr_size_c < vector_array_size`
 - în această situație, matricea de intrare va ocupa mai multe linii din memoria locală și nu va mai fi nevoie să stocăm vectorul de intrare de mai multe ori, ci îl vom refolosi pentru fiecare înmulțire cu un set de coloane, după cum este ilustrat în Figura 4.5b. Din motive de aliniere, vom stoca doar coloane întregi într-o linie de procesare și, dacă aceasta va fi incompletă, vom adăuga un *padding* până la completarea ei. Poate apărea situația în care ultima dintre liniile din LS în care este stocată matricea este incompletă, cum este cazul liniei $LS[i]$ din figură, care trebuie tratată separat, în toate celelalte linii fiind stocate câte k coloane din matricea de intrare. Am considerat că matricea este stocată pe x linii din LS, iar vectorii ocupă, în total, y linii din LS.

Având în vedere notațiile menționate, se vor parcurge următorii pași, sintetizați și în pseudocodul din Listarea 4.2:

1. Avem y iterații pentru fiecare linie din LS care conține vectori, la începutul căreia se aduce câte una dintre ele într-un registru. În interiorul fiecărei iterații:
 - 1.1. Efectuăm $x - 1$ iterații pentru fiecare linie (cu excepția ultimei) din LS în care se află stocată matricea, la începutul căreia aceasta se transferă într-un registru. În interiorul fiecărei iterații:
 - Efectuăm înmulțirea dintre registrele din fiecare PE descrisă în Secțiunea 4.2.
 - Vom realiza câte două reducții pentru fiecare dintre cele k coloane dintr-o linie de procesare.
 - 1.2. Ultima linie din LS care conține o parte din matrice va fi tratată separat, deoarece este posibil să se efectueze un număr mai mic de reducții. De exemplu, în Figura 4.5b se vor efectua reducții doar pe două blocuri după ce, în prealabil, s-au înmulțit registrele ca într-un caz obișnuit. Acest pas este necesar pentru a nu procesa elemente în plus, pe care ar trebui apoi să le luăm în calcul la citirea rezultatelor.
2. Incrementăm registrul folosit în adresarea liniilor din LS ce conțin vectori și trecem la iterația pentru următoarea linie care conține un vector de intrare.


```

1 for i = 1 to y:
2   load input array from LS[i];
3   multiply registers;
4   for j = 1 to arr_size_c * k:
5     select block to reduce;
6     reduce;

```

Listarea 4.1: Pseudocod pentru înmulțirea dintre un vector și o matrice atunci când cel puțin o matrice poate fi scrisă într-o linie din memoria locală

```

1 for i = 1 to y:
2   load input array from LS[x+i-1];
3   for j = 1 to x-1:
4     load matrix block from LS[j-1];
5     multiply registers;
6     for jj = 1 to k:
7       select block to reduce;
8       reduce;
9
10  load last matrix block from LS[x-1];
11  multiply registers;
12  for all partial blocks to reduce:
13    select block to reduce;
14    reduce block;

```

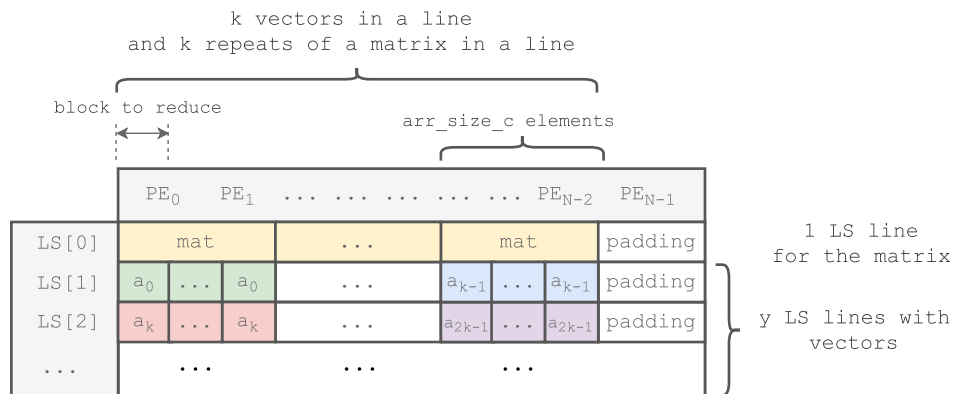
Listarea 4.2: Pseudocod pentru înmulțirea dintre un vector și o matrice atunci când cel puțin o coloană poate fi scrisă într-o linie din memoria locală

```

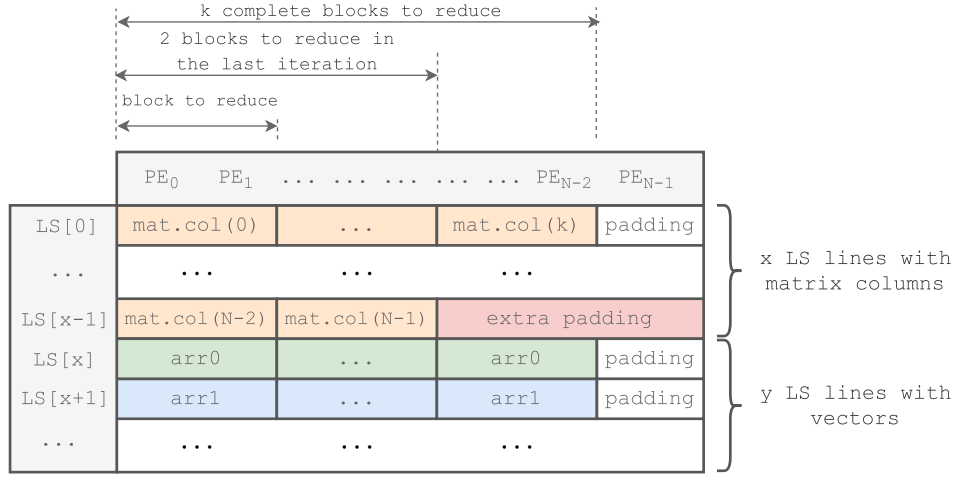
1 for i = 1 to y:
2   for ii = 1 to N:
3     reset the array pointer ARR_PTR = x + (i-1) * n;
4     for j = 1 to n:
5       load current matrix block from LS[(j-1)*n];
6       load current array chunk from LS[ARR_PTR];
7       multiply registers;
8       reduce processing line;
9       ARR_PTR++;
10  ARR_PTR++;

```

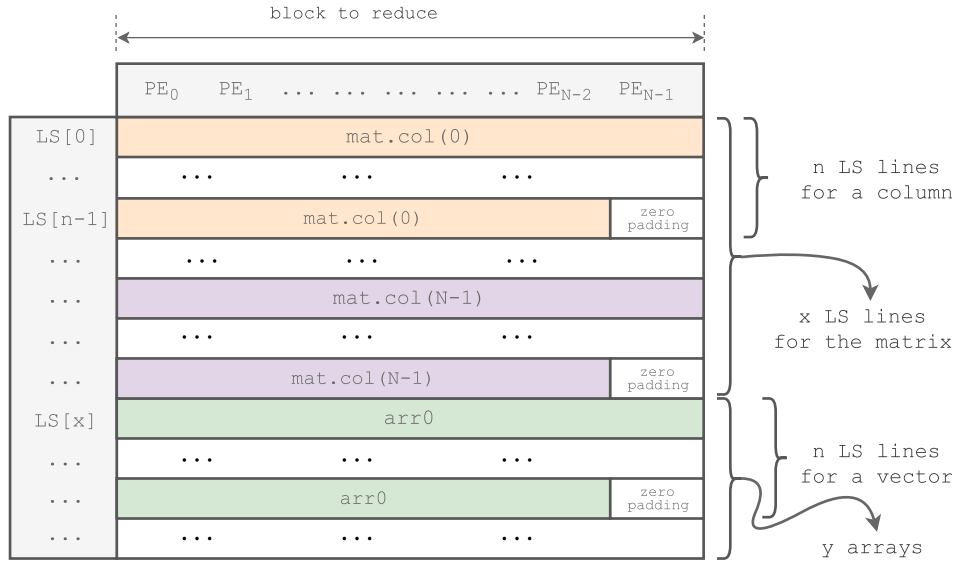
Listarea 4.3: Pseudocod pentru înmulțirea dintre un vector și o matrice atunci când o coloană din matrice ocupă mai multe linii din memoria locală



(a) Cazul în care o linie din LS conține cel puțin o matrice (sistem cu 2 - 8 antene).



(b) Cazul în care o linie din LS conține cel puțin o coloană din matricea de intrare (sistem cu 9 - 64 antene).



(c) Cazul în care o coloană dintr-o matrice este stocată pe mai multe linii din LS (sistem cu mai mult de 64 antene).

Figura 4.5: Organizarea elementelor în memoria locală a acceleratorului în cazul înmulțirii dintre un vector linie și o matrice.

4.4 Kernel pentru produsul înlănțuit dintre un vector linie, o matrice pătratică și un vector coloană

În secțiunea precedentă a fost descris un algoritm pentru realizarea înmulțirii dintre un vector linie și o matrice pătratică, adică prima parte a înmulțirii de la numitorul Ecuației (4.6), notată $X_M \triangleq \mathbf{a}^H(\theta) \mathbf{V}_{N_{sq}}$. Dorim acum să găsim un algoritm pentru a realiza întregul produs $\mathbf{a}^H(\theta) \mathbf{V}_{N_{sq}} \mathbf{a}(\theta)$. Elaborând pe baza relațiilor obținute anterior, avem:

$$X_M \mathbf{a}(\theta) = \begin{bmatrix} \sum_{i=0}^{M-1} \bar{a}_i v_{i,0} & \sum_{i=0}^{M-1} \bar{a}_i v_{i,1} & \dots & \sum_{i=0}^{M-1} \bar{a}_i v_{i,M-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{M-1} \end{bmatrix} \quad (4.13)$$

Deducem că întregul produs se poate scrie și sub forma:

$$P_{MUSIC}(\theta)^{-1} = \left[a_0 \sum_{i=0}^{M-1} \bar{a}_i v_{i,0} + a_1 \sum_{i=0}^{M-1} \bar{a}_i v_{i,1} + \dots + a_{M-1} \sum_{i=0}^{M-1} \bar{a}_i v_{i,M-1} \right] \quad (4.14)$$

Putem scrie una din sumele de produse sub forma:

$$a_k \sum_{i=0}^{M-1} \bar{a}_i v_{i,k} = a_k \left(\sum_{i=0}^{M-1} (x_i x_{i,k} - \bar{y}_i y_{i,k}) + j \sum_{i=0}^{M-1} (x_i y_{i,k} + \bar{y}_i x_{i,k}) \right) \quad (4.15)$$

Notăm

$$E_{i,k}^{(1)} \stackrel{not}{=} x_i x_{i,k} - \bar{y}_i y_{i,k}, \quad i = \overline{0, M-1}, k = \overline{0, M-1} \quad (4.16)$$

$$E_{i,k}^{(2)} \stackrel{not}{=} x_i y_{i,k} + \bar{y}_i x_{i,k}, \quad i = \overline{0, M-1}, k = \overline{0, M-1} \quad (4.17)$$

și putem scrie:

$$a_k \sum_{i=0}^{M-1} \bar{a}_i v_{i,k} = (x_k + jy_k) \left(\sum_{i=0}^{M-1} E_{i,k}^{(1)} + j \sum_{i=0}^{M-1} E_{i,k}^{(2)} \right) \quad (4.18)$$

$$a_k \sum_{i=0}^{M-1} \bar{a}_i v_{i,k} = \left(x_k \sum_{i=0}^{M-1} E_{i,k}^{(1)} - y_k \sum_{i=0}^{M-1} E_{i,k}^{(2)} \right) + j \left(x_k \sum_{i=0}^{M-1} E_{i,k}^{(2)} + y_k \sum_{i=0}^{M-1} E_{i,k}^{(1)} \right) \quad (4.19)$$

În Secțiunea 4.2, unde am detaliat înmulțirea a doi vectori de numere complexe, am considerat aranjamentul datelor din registrele acceleratorului din Figura 4.2, iar cu notațiile făcute observăm că produsele componente ale expresiei $E_{i,k}^{(1)}$ se găsesc în registrele R3, iar produsele componente ale expresiei $E_{i,k}^{(2)}$ au fost reținute în registrele R4, unde $k = \overline{0, M-1}$ reprezintă coloana din matricea \mathbf{V}_{Ns_q} pentru care au fost calculate. Rezultă că, dacă înmulțim partea reală a unui element a_k din vectorul $a(\theta)$ cu fiecare produs component al expresiei $E_{i,k}^{(1)}$ corespunzătoare unei coloane k și, similar, partea imaginară (cu semn schimbat) a aceluiași element cu fiecare produs din expresia $E_{i,k}^{(2)}$ și sumăm produsele astfel obținute din toate coloanele, vom obține o valoare a spectrului MUSIC.

Aranjamentul propus, particularizat pentru doi vectori de lungime egală cu 4 și o matrice pătratică de dimensiune 4×4 , este ilustrat în figura 4.6. Partea imaginară a unui număr este marcată printr-o figură hașurată și se observă cum părțile reale și imaginare sunt repetate pe lungimea unei coloane a matricei pentru a putea fi obținute produsele anterior menționate.

În cazul unor serii de astfel de înmulțiri înlănțuite care folosesc aceeași matrice de intrare (cum este cazul în algoritmul MUSIC), aranjarea elementelor vectorilor se va face în mod similar cu cazul înmulțirii dintre un vector linie și o matrice din Figura 4.5, modificarea constând în faptul că în memorie vom avea trei zone: una pentru elementele vectorului linie, una pentru părțile reale ale vectorului coloană și una pentru părțile imaginare ale acestuia. Prin urmare, un dezavantaj al acestei metode va consta în faptul că vom putea cuprinde mai puține elemente de intrare într-o execuție decât în cazul precedent.

În cazul în care `mat_size_c` \leq `vector_array_size`, cele trei zone de memorie vor fi egale. Dacă, însă, `mat_size_c` $>$ `vector_array_size`, fiecare vector linie va trebui stocat o singură dată, dar elementele vectorului coloană vor ocupa un număr egal de linii din

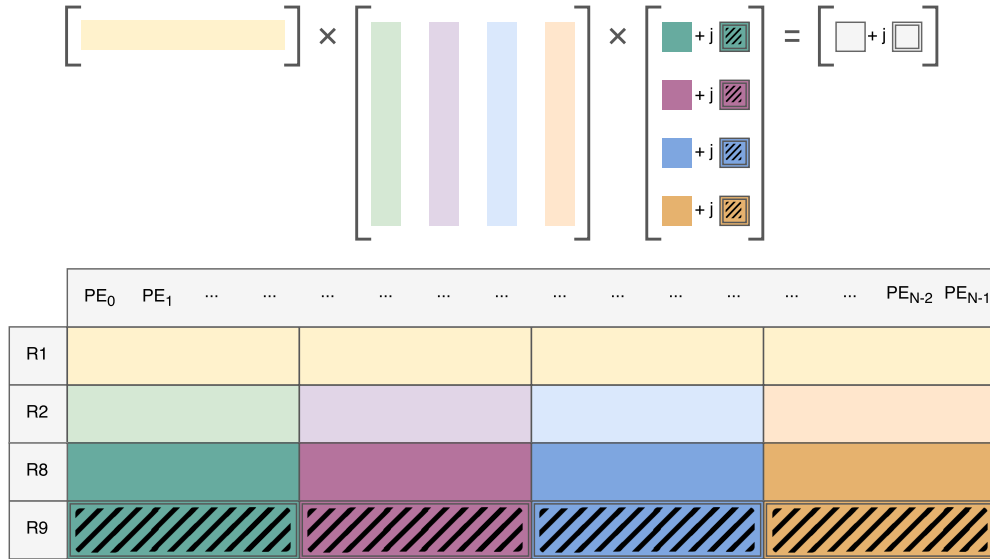


Figura 4.6: Aranjarea elementelor vectorilor în ConnexArray în cazul înmulțirii în lanțuite a unui vector, o matrice și a altui vector

LS ca și matricea de intrare. În Anexa 2.3 se găsește implementarea kernelului pentru prima situație descrisă și o comparație între cele două metode va fi realizată în Capitolul 5, Secțiunea 5.3.

4.5 Kernel pentru autocorelația unui semnal

Metoda de realizare a autocorelației a fost descrisă în Secțiunea 3.2.5, conceptul de bază fiind faptul că autocorelația este estimată prin intermediul eşantioanelor semnalului de intrare care ajunge la cele N antene, efectuată pe o „captură” de K eşantioane. Ca pas opțional, se poate aplica și o mediere antegradă-retrogradă pentru o acuratețe mai bună a estimării.

Primul pas este estimarea matricei de autocorelație \mathbf{C}_X din matricea formată din cele K eşantioane de la cele N semnale de intrare:

$$\mathbf{C}_X = \frac{1}{K} \mathbf{X} \mathbf{X}^H. \quad (4.20)$$

Estimatul matricei de autocorelație devine:

$$\mathbf{X} \triangleq \begin{bmatrix} x_{0,0} & x_{0,1} & \dots & x_{0,K-1} \\ x_{1,0} & x_{1,1} & \dots & x_{1,K-1} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N-1,0} & x_{N-1,1} & \dots & x_{N-1,K-1} \end{bmatrix} \quad (4.21)$$

$$\mathbf{C}_X = \mathbf{X}\mathbf{X}^H = \begin{bmatrix} \sum_{n=0, K-1} x_{0,n} x_{n,0}^* & \sum_{n=0, K-1} x_{0,n} x_{n,1}^* & \cdots & \sum_{n=0, K-1} x_{0,n} x_{n, N-1}^* \\ \sum_{n=0, K-1} x_{1,n} x_{n,0}^* & \sum_{n=0, K-1} x_{1,n} x_{n,1}^* & \cdots & \sum_{n=0, K-1} x_{1,n} x_{n, N-1}^* \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{n=0, K-1} x_{N-1,n} x_{n,0}^* & \sum_{n=0, K-1} x_{N-1,n} x_{n,1}^* & \cdots & \sum_{n=0, K-1} x_{N-1,n} x_{n, N-1}^* \end{bmatrix} \quad (4.22)$$

unde

$$x_{m,n} = a_{m,n} + j b_{m,n}, \quad m = \overline{0, M-1}, n = \overline{0, M-1} \quad (4.23)$$

Deci, un element $c_{m,n}$ al matricei \mathbf{C}_X este:

$$c_{m,n} = \sum_{k=0}^{K-1} (a_{m,k} a_{k,n} + b_{m,k} b_{k,n}) + j \sum_{k=0}^{K-1} (-a_{m,k} b_{k,n} + b_{m,k} a_{k,n}) \quad (4.24)$$

Observăm că \mathbf{C}_X este o matrice Hermitică, deci $c_{ij} = \overline{c_{ji}}$, unde c_{ij} este un element al matricei și $i = \overline{0, N-1}$, $j = \overline{0, N-1}$, ceea ce înseamnă că va fi de ajuns să calculăm doar elementele de pe diagonala principală și de deasupra ei.

În plus, se observă că expresia unui element de-al matricei de autocorelației este foarte asemănătoare cu cea obținută în Ecuația (4.12), deci algoritmul precedent va putea fi aplicat cu modificări minime.

Dorim să avem în fiecare execuție a kernelului un număr cât mai mare de date de procesat, prin urmare, vom încerca să realizăm întreaga înmulțire de matrice într-o singură execuție. Deși autocorelația se realizează între o matrice și hermitica ei, vom prefera să încărcăm doar matricea de intrare în forma inițială în memoria acceleratorului, precum în Figura 4.7 și să adaptăm calculul înmulțirii, astfel încât nu se mai inversează semnul produselor $b_{m,k} b_{k,n}$, ci semnul produselor de forma $a_{m,k} b_{k,n}$, unde $k = \overline{0, K-1}$. Listarea 4.4 conține un pseudocod pentru modul de lucru descris pentru kernel, iar în Anexa 2.4 se găsește și implementarea acestuia.

```

1 for i = 1 to N
2   load LS[i];
3   for j = i to N
4     load LS[j];
5     multiply registers and invert signs;
6     reduce registers;
```

Listarea 4.4: Pseudocod pentru kernel care realizează autocorelația unui semnal

În cazul de față, memoria locală are o capacitate de $1024 \cdot 128$ elemente de 16 biți, iar numărul de elemente necesare pentru a încărca matricea este $2 \cdot N \cdot K$, deoarece părțile reale și cele imaginare ale datelor de intrare sunt considerate elemente distincte. Prin urmare, avem o flexibilitate destul de mare în alegerea configurației sistemului. Pentru un sistem ce conține 64 antene, de exemplu, putem avea o dimensiune maximă a capturii de 1024 elemente, care oferă o acuratețe satisfăcătoare a rezultatului final de aproximativ 0.15° . Dacă se dorește folosirea unui număr mai mare de antene și o estimare mai imprecisă a autocorelației nu este acceptabilă, procesarea poate fi, și în acest caz, împărțită în blocuri de procesare egale, încărcând liniile necesare astfel încât să obținem toate produsele dorite.

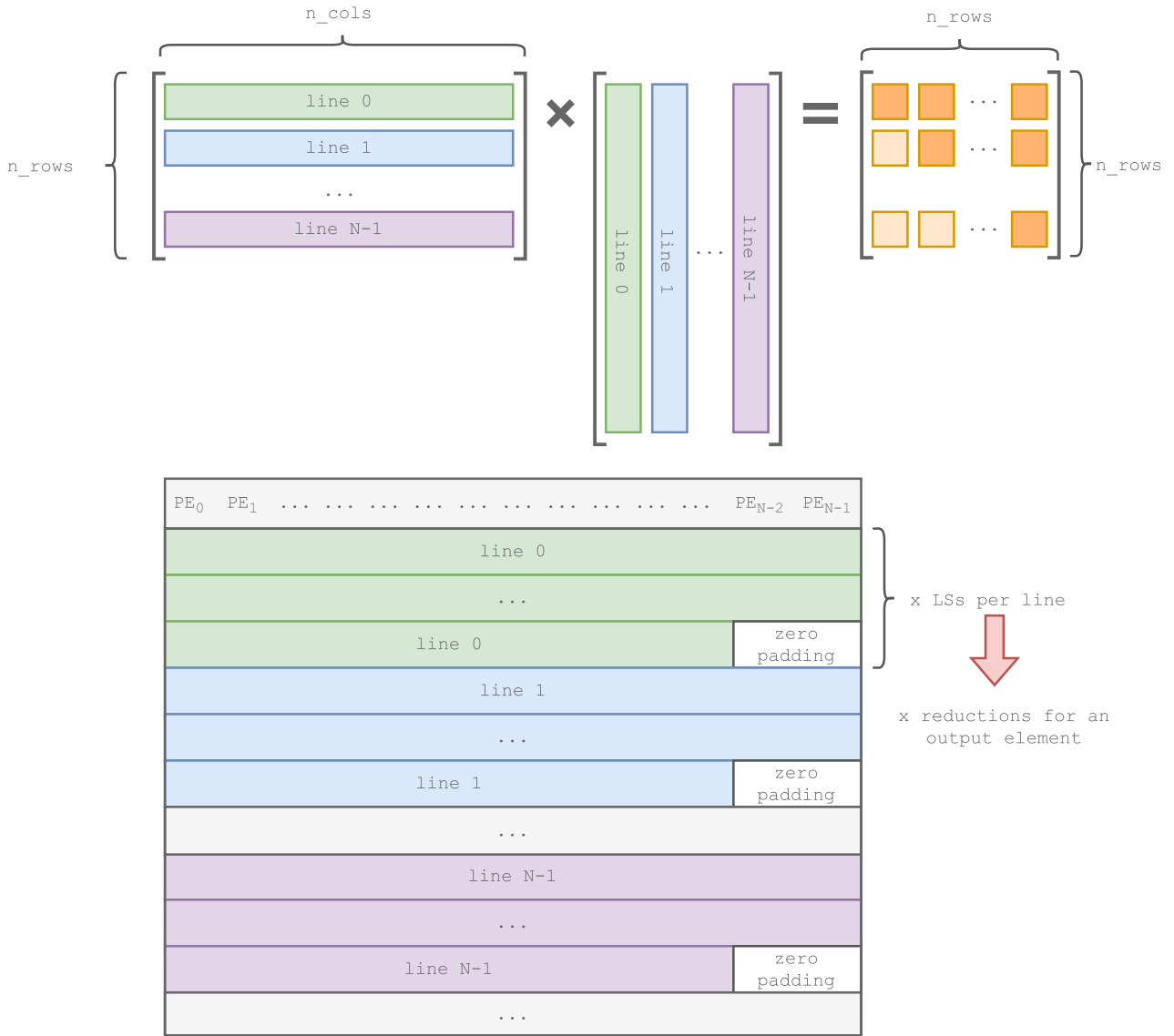


Figura 4.7: Aranjarea datelor de intrare în acceleratorul ConnexArray pentru un kernel care calculează autocorelația unui semnal

4.6 Integrarea kernelurilor în GNU Radio

Întrucât kernelurile au o funcționalitate generică, putând fi folosite și pentru alți algoritmi de procesare de semnale, am dorit integrarea lor în blocuri dintr-un modul OOT în GNU Radio, pentru a fi independente de lanțul de procesare pentru algoritmul MUSIC ales.

Este important de avut în vedere faptul că blocurile vor trebui să comunice cu cozile acceleratorului, deci vor avea nevoie de căile către acestea pentru a putea crea o instanță a obiectului `ConnexMachine` dependentă de acceleratorul folosit, pe care le vor primi ca parametri de intrare odată cu crearea blocului. Programul principal, care instanțiază blocul, va primi, la rândul său, aceste căi ca argumente în linia de comandă. Vom folosi un script pentru lansarea sa în execuție, în care putem specifica dacă dorim să folosim simulatorul sau echipamentul hardware, programul în sine fiind agnostic în legătură cu acest aspect. Diferența va consta doar în căile primite ca argument. În continuare, vor fi descrise specificațiile blocurilor pentru fiecare kernel creat.

4.6.1 Bloc GNU Radio pentru realizarea înmulțirii dintre o serie de vectori linie și o matrice

Deoarece cazul în care vom avea de înmulțit un singur vector linie cu o singură matrice este simplu de efectuat și deoarece există situații în care va fi necesar să calculăm produsul dintre mai mulți vectori și aceeași matrice (cum este cazul în algoritmul MUSIC), am preferat pentru implementare cea de-a doua variantă.

Am creat un bloc de sine stătător care are două intrări și o ieșire de tip vectori de numere complexe, ceea ce înseamnă că elementele nu sunt procesate unul câte unul, ci se așteaptă colectarea unui număr suficient de elemente de intrare pentru a putea începe procesarea unui element de ieșire, care este la rândul său de tip vector de numere complexe. Specificațiile intrărilor și ieșirii sunt următoarele, ilustrate și în Figura 4.8:

- Prima intrare - primește blocuri de câte `nr_arrays` vectori de lungime `arr_size`, de tip de date complex.
- A doua intrare - primește o matrice pătratică de dimensiune `arr_size x arr_size`, stocată în memorie coloană cu coloană.
- Ieșire - rezultatul înmulțirii dintre cele `nr_arrays` vectori de intrare și matrice. Din moment ce rezultatul înmulțirii dintre un singur vector și o matrice conține `arr_size` elemente, ieșirea va avea în total un număr de `arr_size x nr_arrays` elemente.

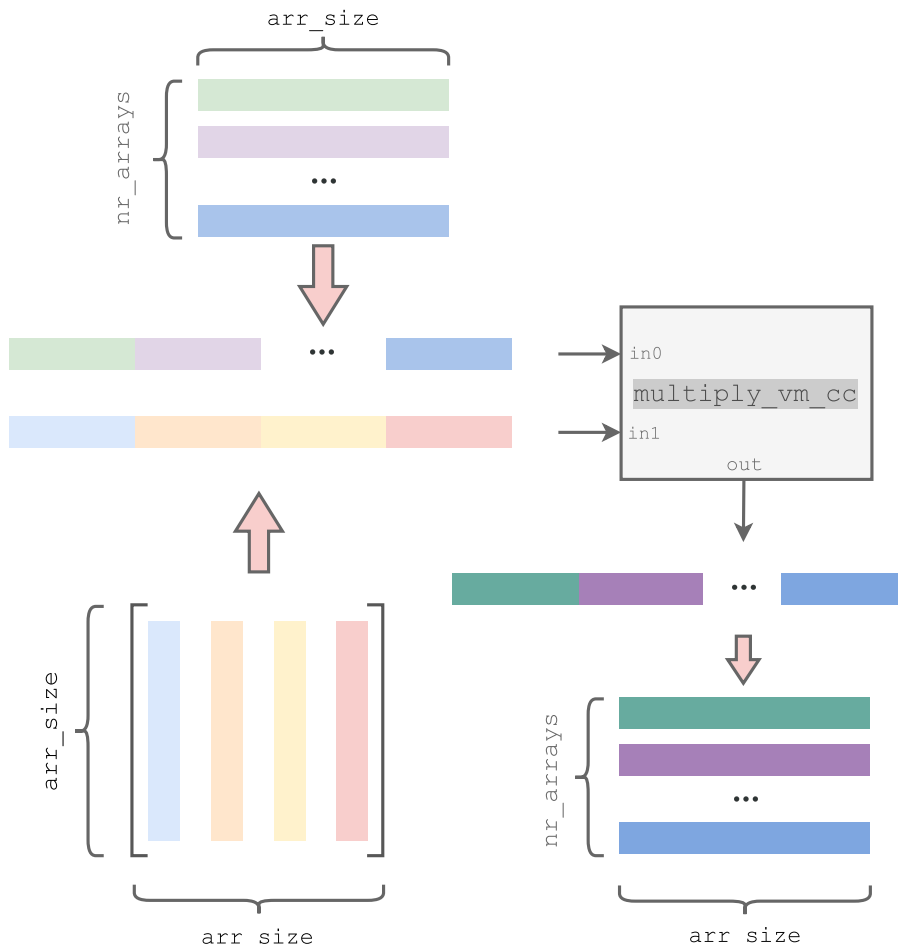


Figura 4.8: Structura intrărilor și ieșirilor blocului GNU Radio care realizează înmulțirea dintre o serie de vectori de intrare și aceeași matrice

Deoarece există relații diferite între numărul de elemente de intrare și cele de ieșire necesare, blocul va fi de tip `general` și va folosi o metodă `forecast` pentru a le asigura. Întrucât acceleratorul ConnexArray are o capacitate de `vector_array_size = 128` PE-uri și numărul de vectori pe prima intrare poate fi destul de mare, vom separa elementele de intrare în blocuri care vor fi procesate succesiv.

Înainte de fiecare lansare în execuție a unui kernel, datele de intrare trebuie să fie „pregătite”, adică scalate și salvate într-un vector de elemente de tip întreg, fără semn, pe 16 biți. Matricea de intrare poate fi pregătită o singură dată pentru fiecare element de ieșire, în timp ce pentru vectorii de intrare acest lucru trebuie să se facă la începutul fiecărui bloc.

După ce avem datele de intrare pregătite, putem să lansăm în execuție kernelul, operație care nu este blocantă, deci firul de execuție principal își continuă parcursul. Obținem datele de ieșire prin citirea unui număr de elemente din coada de reducere. Spre deosebire de lansarea în execuție, citirea datelor din coadă *este* blocantă, deoarece trebuie să avem certitudinea că citim datele corecte. Odată ce ele sunt gata, trebuie scalate înapoi și convertite la tipul de date complex cu părțile reală și imaginară în virgulă mobilă, folosit pentru datele de ieșire.

Decizia în privința procesării care se va efectua în paralel va fi luată în capitolul 5, deoarece trebuie să realizăm un profil al programului pentru a stabili care sunt punctele critice ale programului ce ar putea beneficia de un mod de lucru distribuit.

4.6.2 Bloc GNU Radio pentru calcularea spectrului MUSIC

În calculul spectrului MUSIC intervine, la numitorul acestuia, o înmulțire înlănțuită dintre un vector linie, o matrice pătratică și un vector coloană (transpusul și conjugatul vectorului linie) care, după cum a fost deja menționat, lasă deschise cel puțin două moduri de abordare: fie vom realiza prima parte a înmulțirii, cea dintre un vector linie și o matrice pătratică, pe acceleratorul ConnexArray, iar rezultatul va fi înmulțit cu ultimul vector coloană pe procesorul gazdă ARM, fie vom realiza întreaga înmulțire pe procesorul ConnexArray.

În ambele cazuri, blocul GNU Radio pentru calcularea spectrului MUSIC primește la intrare o matrice pătratică vectorizată ce reprezintă estimatul autocorelației semnalului de intrare, care va fi descompus în vectori proprii ce vor interveni în calculul spectrului MUSIC. În continuare, procesarea se face în mod similar cu cea a blocului descris anterior, împărțind datele de intrare în blocuri de procesare a căror dimensiune va fi stabilită în funcție de metoda folosită, astfel încât memoria să fie ocupată la capacitate maximă într-o execuție. Diferența va consta în faptul că vectorii care apar în înmulțire pot fi pregătiți pentru procesarea pe accelerator o singură dată, în constructorul blocului, eliminând, astfel, o parte importantă din punct de vedere computațional. Matricea va fi pregătită pentru fiecare element de ieșire, ea fiind comună tuturor înmulțirilor pentru un calcul al spectrului MUSIC.

În testarea acestui modul, au fost observate câteva aspecte importante de menționat:

- În cazul primei variante, în care doar prima înmulțire dintre un vector linie și o matrice este realizată pe accelerator, am constatat că obținem o precizie medie a rezultatelor la ieșirea din accelerator de ordinul 10^{-4} . În punctele spectrului MUSIC corespunzătoare

unghiului de incidență al semnalului căutat, în teorie, numitorul expresiei spectrului ar trebui să fie cât mai apropiat de zero, iar spectrul MUSIC să aibă o valoare cât mai mare. În practică, acestea au valori de aproximativ $10^{-5} - 10^{-6}$ sau poate chiar mai mici, iar valorile din rezultatul înmulțirii intermediare sunt de ordinul $10^{-4} - 10^{-5}$, dar din cauza faptului că precizia obținută cu folosirea datelor în virgulă fixă pe accelerator este de aproximativ 10^{-4} , erorile în punctele de interes vor fi semnificative. Acest lucru conduce, în unele cazuri, la apariția mai multor maxime foarte apropiate în jurul unuia dintre unghiuri, care vor fi alese ambele drept unghiuri de incidență, în detrimentul unuia dintre unghiurile de incidență corecte. În Figura 4.9a este redată reprezentarea corectă a spectrului MUSIC, iar în Figura 4.9b este prezentată situația incorectă cauzată de precizia insuficientă. O soluție în acest sens ar fi modificarea algoritmului de căutare a maximelor locale ale spectrului MUSIC, astfel încât să ignore două puncte foarte apropiate și să caute altul, dacă acesta există. Acest lucru nu ar afecta neapărat rezoluția algoritmului, deoarece el nu poate distinge oricum între două surse de semnal foarte apropiate, dar va introduce o procesare în plus în blocul de căutare.

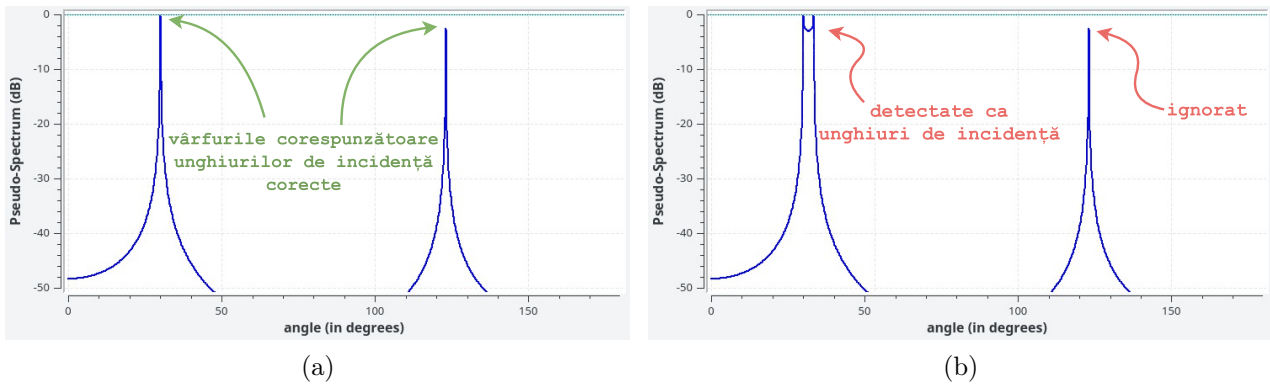


Figura 4.9: Reprezentările corectă și incorectă ale spectrului MUSIC

- În cazul în care folosim înmulțirea înlănțuită executată integral pe accelerator, datele vor avea de suferit, din nou, din cauza preciziei. Din cauza faptului că ultima înmulțire este realizată, și ea, pe accelerator, rezultatele cu ordin mai mic de $10^{-7} - 10^{-8}$ vor fi direct approximate cu zero, ce conduce la o valoare infinită a spectrului în acele puncte. În plus, în implementarea de față, spectrul MUSIC este normat la valoarea sa maximă și apoi logaritmat. Dacă în spectru se găsește cel puțin o valoare de ∞ , prin normarea tuturor celorlalte valori la aceasta se va obține rezultatul zero care, logaritmat, va conduce la apariția unor valori de $-\infty$ în tot spectrul, fără a mai putea distinge punctele de interes. Cu alte cuvinte, implementarea de față nu este adaptată lucrului cu valori infinite ale spectrului MUSIC. O soluție temporară este înlocuirea valorilor nule cu unele nenule, pentru a evita situația descrisă, dar o soluție de viitor ar putea lua în calcul tratarea punctelor infinite din spectru ca fiind valide.

4.6.3 Bloc GNU Radio pentru realizarea autocorelației

Acest bloc are modificări minime față de cel precedent, în ceea ce privește intrările și modul de parcurgere al elementelor, ilustrat în Figura 4.10. Avem, în acest caz, o singură intrare care reprezintă o matrice de `nr_arrays` \times `array_size` elemente vectorizată, unde numărul de linii este echivalentul numărului de antene din sistem, iar numărul de coloane reprezintă dimensiunea capturii pe care se realizează estimatul autocorelației. Ieșirea reprezintă matricea de autocorelație estimată. După cum am menționat, aceasta este o matrice Hermitică, ceea

ce înseamnă că elementele simetrice față de diagonala principală sunt conjugate, deci nu va fi nevoie să calculăm decât $\text{nr_arrays}(\text{nr_arrays} + 1) / 2$ elemente de deasupra diagonalei principale (inclusiv) din totalul de $\text{nr_arrays} * \text{nr_arrays}$ și doar să inversăm semnul părții imaginare pentru restul.



Figura 4.10: Structura intrărilor și ieșirilor blocului GNU Radio care realizează autocorelația

În programul principal, putem să citim un număr de reducere necesar formării unui element de ieșire și să procesăm acest element de ieșire în timpul în care se calculează următorul, efectuând, în acest mod, o ușoară paralelizare.

Acest bloc nu este atât de sensibil la erorile de precizie, atingând cu el același nivel de acuratețe a estimării unghiului de incidență ca și în cazul implementării originale.

Capitolul 5

Evaluarea performanțelor obținute

Acest capitol își propune prezentarea performanțelor obținute cu ajutorul kernelurilor pentru procesorul ConnexArray, care au fost integrate în două blocuri de procesare din implementarea algoritmului MUSIC, și anume: blocul care calculează un estimat al autocorelației semnalului de intrare și blocul care calculează spectrul MUSIC. Astfel, în Secțiunea 5.1 vom prezenta modalitatea de evaluare a performanțelor și de comparare a acestora cu cea a blocurilor din implementarea originală. Se prezintă, apoi, rezultatele obținute pentru cele două blocuri enumerate anterior în Secțiunile 5.2 și, respectiv, 5.3, respectând ordinea în care acestea apar în lanțul de procesare MUSIC. Secțiunea 5.4 urmărește rezultatele blocurilor atunci când acestea sunt integrate în întregul lanț de procesare și, în final, în Secțiunea 5.5 stabilim concluzii și aspecte ce ar putea fi îmbunătățite pe viitor.

5.1 Metodologia de evaluare a performanțelor

Modul de construire și execuție a lanțului de procesare, precum și utilitățile folosite au fost detaliate în Secțiunea 3.3.1, fiind păstrate și în evaluarea kernelurilor ConnexArray. Compararea performanțelor blocurilor de sine stătătoare sau a întregului lanț de procesare se va face operând pe același set de date de intrare citite dintr-un fișier și comparând timpul în care acestea finalizează procesarea.

Este important de notat faptul că ne interesează timpul petrecut în execuția programului, în termenii percepției umane, denumit și *wall-clock time* sau *wall-time*, și nu timpul petrecut pe procesor (*CPU time*). Timpul petrecut pe procesor se referă la durata care a fost petrecută procesând date efectiv pe procesor, fără a lua în considerare și timpul de așteptare după datele de intrare sau cel necesar pentru sincronizarea între mai multe fire de execuție. Având în vedere că GNU Radio poate lansa mai multe fire de execuție pentru procesare, care pot avea anumite dependențe de sincronizare, este posibil ca timpul petrecut pe procesor să nu reflecte realitatea cu acuratețe, motiv pentru care vom compara timpul total de execuție a programelor.

Putem măsura timpul total de execuție cu metoda prezentată în Anexa 3.1 (valabilă doar pentru standardul C++11), cu o precizie de ordinul milisecundelor, care este suficientă pentru datele folosite în cazul de față. Deoarece acesta poate să difere de la o execuție la alta din diverse motive, precum imprecizia metodei care măsoară timpul sau evenimente hardware care pot modifica timpul de acces la date, pentru fiecare caz analizat s-a efectuat un număr de 10

măsurători și s-a făcut o medie a timpului obținut.

Toate rezultatele au fost obținute pe placa de dezvoltare Xilinx ZedBoard Zynq-7000, pe al cărei chip FPGA se află implementat acceleratorul ConnexArray, cu o dimensiune a liniilor de procesare de 256 B, care include 128 elemente de procesare și o memorie locală de 256 KB.

5.2 Evaluarea performanțelor blocului ce calculează autocorelația semnalului de intrare

5.2.1 Procesare liniară

În forma sa descrisă în Secțiunea 4.6.3, blocul de autocorelație nu realizează decât o mică paralelizare în prelucrarea datelor de ieșire din accelerator, comportarea sa fiind, în mare parte, liniară. Realizăm un profil conform metodologiei descrise în secțiunea anterioară pentru un sistem cu următoarea configurație:

- Număr antene: 4
- Număr semnale de intrare: 2
- Dimensiunea spectrului MUSIC: 1024
- Dimensiunea capturii pentru autocorelație: 2048
- Intervalul de suprapunere dintre capturi: 512

Overhead	Command	Symbol
43,67%	autocorrelate_cnx_impl	gr::doa::autocorrelate_cnx_impl::prepareInData
9,75%	autocorrelate_cnx_impl	std::complex<float>::imag[abi:cxx11]
9,53%	autocorrelate_cnx_impl	std::complex<float>::real[abi:cxx11]
1,89%	autocorrelate_cnx_impl	__copy_from_user
1,86%	autocorrelate_cnx_impl	__gnu_cxx::new_allocator<unsigned short>::new_allocator
1,64%	autocorrelate_cnx_impl	_raw_spin_unlock_irqrestore

Tabela 5.1: Profil pentru blocul de autocorelație care folosește un kernel ConnexArray

Rezultatele obținute au fost sintetizate în Tabelul 5.1. Un profil realizat asupra blocului original de autocorelație din Tabelul 5.2 pe același set de date cu aceeași configurație ne indică faptul că am reușit să eliminăm o mare parte din punctele critice implicate în înmulțirea de matrice, cu prețul unui *overhead* semnificativ introdus de pregătirea datelor de intrare, care consumă 43,67% din numărul total de cicli. Operațiunile de trimitere și primire a datelor către și din accelerator nu introduc întârzieri semnificative, dar operațiuni de construcție a unui element complex implicate în procesarea datelor de ieșire consumă, cumulat, aproximativ 20% din ciclii de execuție.

În urma acestei analize, constatăm că blocul ar putea beneficia de o procesare distribuită în care pregătirea elementelor pentru următoarea lansare în execuție a unui kernel se face simultan cu procesarea datelor curente.

Overhead	Command	Symbol
30,83%	autocorrelate5	cgemmm_
18,74%	autocorrelate5	arma::eop_core<arma::eop_conj>::apply<arma::Mat<std::complex<float> >, arma::Mat<std::complex<float> > >
12,45%	autocorrelate5	arma::op_strans::apply_mat_noalias<std::complex<float>, arma::Mat<std::complex<float> > >
10,34%	autocorrelate5	std::conj<float>
3,78%	autocorrelate5	std::complex<float>::complex
3,20%	autocorrelate5	memcpy
2,61%	autocorrelate5	std::complex<float>::imag[abi:cxx11]
1,90%	autocorrelate5	std::complex<float>::real[abi:cxx11]

Tabela 5.2: Profil pentru blocul de autocorelație în implementarea originală

5.2.2 Procesare distribuită

Metoda `work` a oricărui bloc GNU Radio realizează procesarea efectivă a acelui bloc și este apelată atunci când există un număr suficient de date de intrare pentru a produce cel puțin o dată de ieșire. În practică, însă, de cele mai multe ori, *scheduler*-ul GNU Radio nu apelează metoda `work` pentru a produce un singur element de ieșire, ci mai multe, lucru care poate fi exploatat în modul de lucru distribuit, prin pregătirea în prealabil a datelor de intrare necesare producerii unui element de ieșire ulterior.

Vom proceda precum în Figura 5.1, și anume:

1. Se pregătesc primele date de intrare, se scriu în memoria locală și se lansează în execuție kernelul.
2. Dacă se poate procesa mai mult de un element de ieșire, se intră în bucla de iterație pentru toate elementele de ieșire cu excepția ultimului.
3. Se lansează un fir de execuție care va pregăti datele de intrare pentru următoarea iterație, le va scrie în memoria locală și va lansa o nouă execuție de kernel pe accelerator. Pregătirea datelor viitoare de intrare se face în paralel cu citirea și procesarea datelor de ieșire din accelerator din firul principal de execuție.
4. În firul secundar de execuție, scrierea datelor și lansarea noii execuții trebuie să se facă doar *după* ce s-a terminat procesarea în programul principal, pentru a nu corupe datele. Acest lucru se poate implementa cu ajutorul unei bariere de sincronizare. Simultan cu aceste două operații se poate face, dacă este cazul, medierea datelor în programul principal.
5. Se unesc cele două fire de execuție și se trece la următoarea iterație.

Blocul de autocorelație are un număr de intrări egal cu numărul de antene al sistemului și, pentru compararea timpului de execuție dintre blocul original și cel care folosește kernelul de autocorelație, am aplicat pe fiecare intrare un fișier cu o dimensiune de 12 MB, deci cu un număr de 1,5 milioane de eșantioane de numere complexe, având părțile reală și imaginară reprezentate în virgulă mobilă, simplă precizie. În Figura 5.2 este ilustrată o comparație între timpul de execuție obținut cu cele două blocuri pentru o captură de 2048 de elemente și o suprapunere de 512 elemente, pentru sisteme cu 4, 8, 16 și 32 antene. În medie, timpul de execuție este de

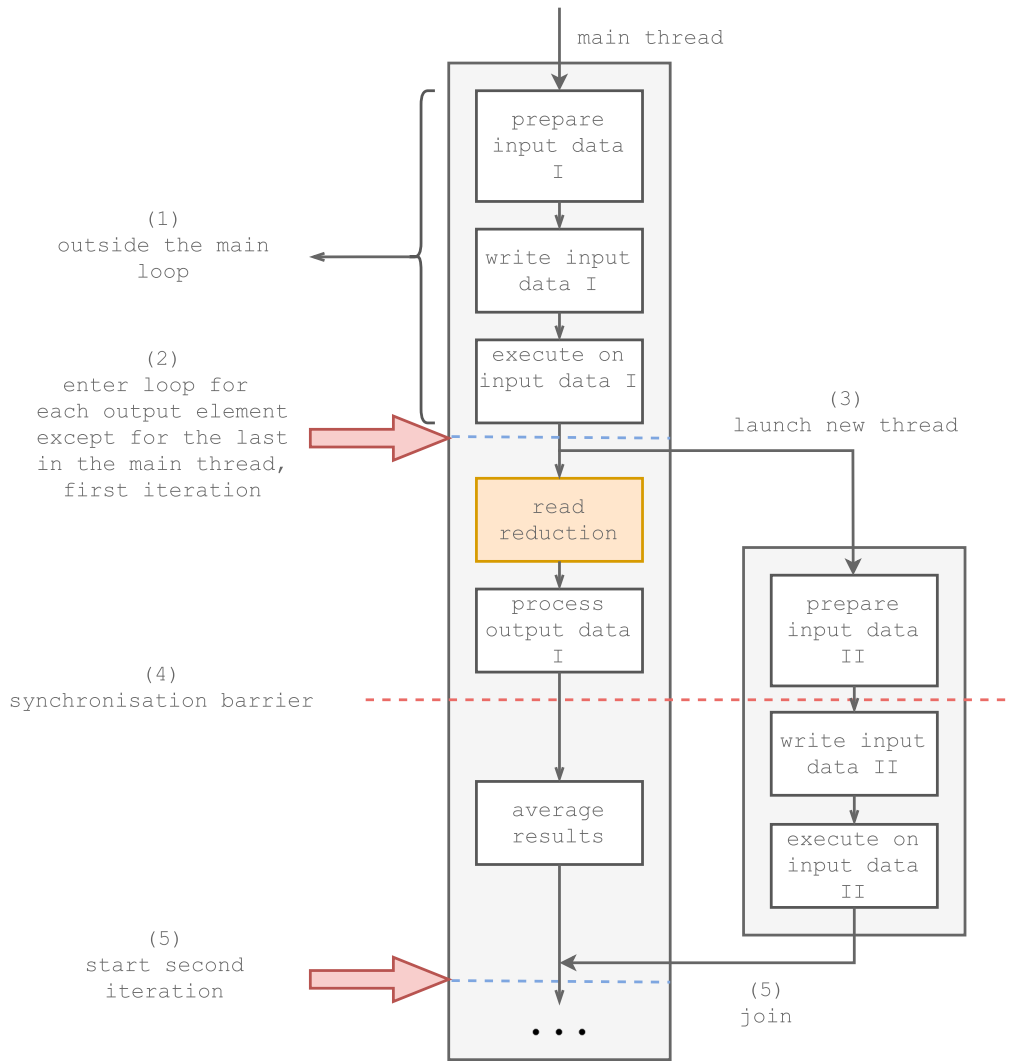


Figura 5.1: Modul de lucru distribuit al blocului de autocorelație

4 ori mai mic decât cel inițial pentru cel puțin 8 antene, dar pentru anumite configurații de antene care permit o mai bună folosire a resurselor, cum este cazul sistemului de antene cu 16 elemente, timpul de execuție poate fi de până la 7 ori mai mic.

După cum a fost precizat în Secțiunea 4.5, dacă dorim să folosim un șir de 64 de antene, putem avea o captură a autocorelației de maxim 1024 de eşantioane. Performanțele obținute de către blocul de autocorelație căruia i-au fost aplicate la intrare fișiere de câte 8 MB (1 milion de eşantioane), care funcționează cu o dimensiune a capturii de 1024 de eşantioane și o suprapunere de 256 de eşantioane pentru sisteme cu 4, 8, 16, 32 și 64 antene, sunt prezentate în Figura 5.3. Folosind până la 8 antene, rezultatele sunt similare cu cele obținute în cazul anterior, dar crescând acest număr, putem obține un timp de execuție de 6 - 7 ori mai mic folosind 16 și 64 antene, sau chiar de aproape 10 ori mai mic pentru un sistem cu 32 de antene. Aceste diferențe se datorează faptului că, fiind necesar un număr mai mic de eşantioane ale semnalului de intrare (1024 față de 2048), ele vor fi disponibile mai rapid și, în majoritatea cazurilor, *scheduler*-ul GNU Radio va apela funcția `work` cu un număr mai mare de date de ieșire pe care le poate produce în acea execuție. Datorită faptului că blocul de autocorelație care folosește kernelul `ConnexArray` pregătește datele de intrare pentru următorul element de ieșire în paralel cu procesarea necesară producerii elementului de ieșire curent, el va beneficia de pe urma apelării funcției `work` cu un număr cât mai mare de elemente de ieșire, motiv

pentru care vom obține performanțe mai bune în cazul în care folosim o captură cu un număr mai mic de eșantioane.

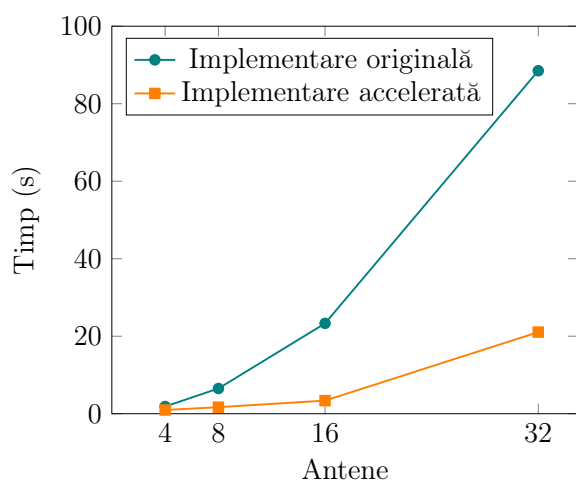


Figura 5.2: Timpul de execuție al blocului de autocorelație pe procesorul ARM pentru o captură de 2048 eșantioane

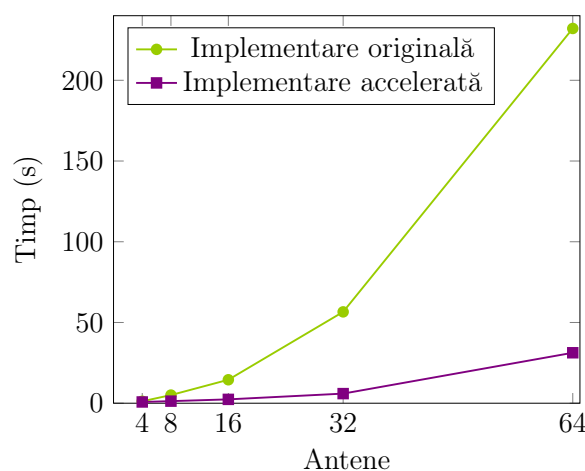


Figura 5.3: Timpul de execuție al blocului de autocorelație pe procesorul ARM pentru o captură de 1024 eșantioane

În privința preciziei, nu s-au observat modificări semnificative datorate scăderii acestui număr de eșantioane, deoarece, prin simulări, s-a constatat că și în acest caz se pot distinge surse de semnal distanțate la o diferență de 1° cu o acuratețe de 0.15° .

În ambele cazuri, performanțele cele mai scăzute se obțin pentru un număr de 4 antene, deoarece capacitatea memoriei acceleratoarelor nu este complet exploatată și nici nu evităm suficient de multe înmulțiri prin calcularea elementelor de deasupra diagonalei principale, inclusiv.

5.3 Evaluarea performanțelor blocului ce calculează spectrul MUSIC

5.3.1 Utilizarea kernelului ce calculează produsul dintre un vector linie și o matrice

Blocul GNU Radio care calculează spectrul MUSIC are o singură intrare, care reprezintă autocorelația semnalului de intrare pe care, în funcție de numărul de antene din sistem, o va interpreta ca pe o matrice liniarizată. În evaluarea performanțelor, am aplicat pe această intrare date care provin dintr-un fișier cu o dimensiune de 8 MB, corespunzătoare a 1 milion de eșantioane de numere fracționare pe 32 de biți. Atunci când sistemul va fi format dintr-un număr mai mare de antene, datele vor fi „consumate” în blocuri mai mici și execuția va fi mai îndelungată. În Tabelul 5.3 sunt prezentate cele mai importante rezultate obținute dintr-un profil al ciclilor de execuție conform metodologiei din Secțiunea 3.3.1, realizat pe acest bloc izolat de lanțul MUSIC, cu următoarea configurație:

- Număr antene: 4
- Număr semnale de intrare: 2
- Dimensiunea spectrului MUSIC: 1024

În acest caz, funcția care pregătește datele la ieșirea din accelerator este cea mai problematică, deoarece se execută în buclă pentru fiecare bloc de date procesat, dar proporția pe care o reprezintă din numărul total de cicli nu este îngrijorătoare. În eventualitatea în care am crea un nou fir de execuție care să se ocupe de această sarcină în paralel cu procesarea unui bloc următor de date, ar trebui să ținem cont și de *overhead*-ul introdus de acesta, care este posibil să compenseze avantajul obținut prin paralelizare. Printre rezultate, încă se găsesc funcții din biblioteca Armadillo, deoarece din înmulțirea înlănțuită dintre un vector linie, o matrice și o coloană, doar primul produs se realizează pe accelerator, rezultând un vector linie, care se înmulțește apoi cu ultimul vector coloană și explică apariția din tabel a simbolului `cgemv_`, corespunzător unei rutine BLAS pentru înmulțirea a doi vectori.

Overhead	Command	Symbol
9,45%	MUSIC_lin_array	gr::doa::MUSIC_lin_array_cnx_impl::prepareOutDataConnex
7,58%	MUSIC_lin_array	arma::Mat<std::complex<float>>::init_warm
7,13%	MUSIC_lin_array	cgemv_
5,75%	MUSIC_lin_array	arma::subview<std::complex<float>>::extract
5,05%	MUSIC_lin_array	__copy_from_user

Tabela 5.3: Profil pentru blocul care calculează spectrul MUSIC cu un kernel ConnexArray

Constatăm, deci, că singurele îmbunătățiri cu un impact mai mare care ar putea fi efectuate sunt legate de folosirea la capacitate maximă a memoriei acceleratorului. Reamintim că aceasta depinde atât de numărul de antene din sistem, cât și de dimensiunea spectrului MUSIC, cea din urmă având impact asupra acurateții măsurătorii. Cu cât se efectuează mai puține înmulțiri într-o execuție a kernelului, cu atât întârzierea introdusă de transferul I/O va deveni mai semnificativă.

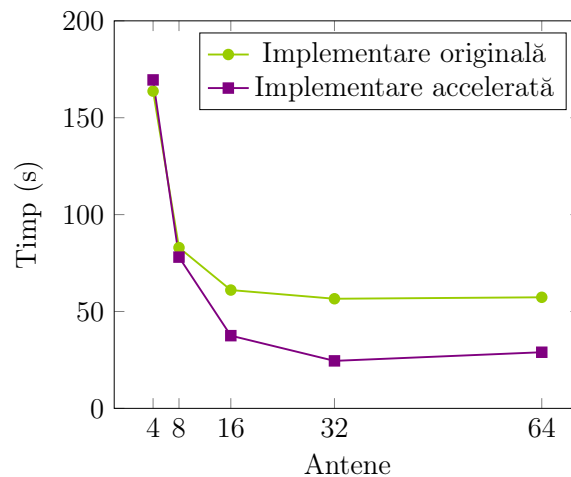


Figura 5.4: Timpul de execuție al blocului ce calculează spectrul MUSIC cu 1024 elemente

Pentru configurația acceleratorului ConnexArray folosită și aranjarea elementelor propusă în Secțiunea 4.3, pentru 4 antene și o dimensiune a spectrului MUSIC de 1024 de elemente, de exemplu, ar fi ocupate doar 256 de linii din LS, deci un sfert din capacitatea sa totală. Atunci când vom calcula numărul de linii din LS disponibile pentru stocarea vectorilor de intrare, dacă avem nu număr mai mic sau egal de 32 de antene, putem să transferăm elementele matricei de intrare în registre înaintea stocării vectorilor, pentru ca aceasta să fie în întregime disponibilă

pentru ei. Pentru mai mult de 32 de antene, nu dispunem de suficiente registre pentru a realiza acest lucru, deci vom folosi varianta inițială.

În acest caz, pentru o dimensiune a spectrului MUSIC de 1024 elemente și pentru configurații de sisteme cu 4, 8, 16, 32 și 64 antene, timpii de execuție obținuți sunt redați în Figura 5.4. Ca și în cazul kernelului pentru autocorelație, pentru un număr mic de antene (de exemplu, 4 sau 8) performanțele sunt similare cu ale blocului inițial care nu folosește acceleratorul. Crescând numărul de antene, însă, este utilizată mai eficient capacitatea de stocare a acceleratorului și timpul de execuție se înjumătățește față de cel al blocului inițial.

5.3.2 Utilizarea kernelului ce calculează produsul înlănțuit dintre un vector linie, o matrice și un vector coloană

În privința kernelului care calculează produsul dintre un vector linie, o matrice și un vector coloană, s-a observat că singura situație în care oferă rezultate mai bune este pentru un număr mic de antene (de exemplu, 4), deoarece pentru dimensiuni mai mari ale șirurilor de antene și, implicit, ale dimensiunii vectorilor, memoria locală a acceleratorului este folosită ineficient, trebuind să ocupăm mai multe linii de stocare pentru a efectua o singură înmulțire. Acest lucru conduce la procesarea în blocuri mai mici și timpul necesar transmiterii datelor pe interfața AXI către accelerator nu mai este neglijabil.

Figura 5.5 prezintă o comparație între timpii de execuție obținuți cu blocul ce calculează spectrul MUSIC folosind un kernel pentru produsul înlănțuit, în aceleași condiții descrise în Secțiunea 5.3.1, și cei prezentați și în Figura 5.4 pentru același bloc în varianta originală și în varianta ce conține kernelul pentru înmulțirea dintre un vector și o matrice.

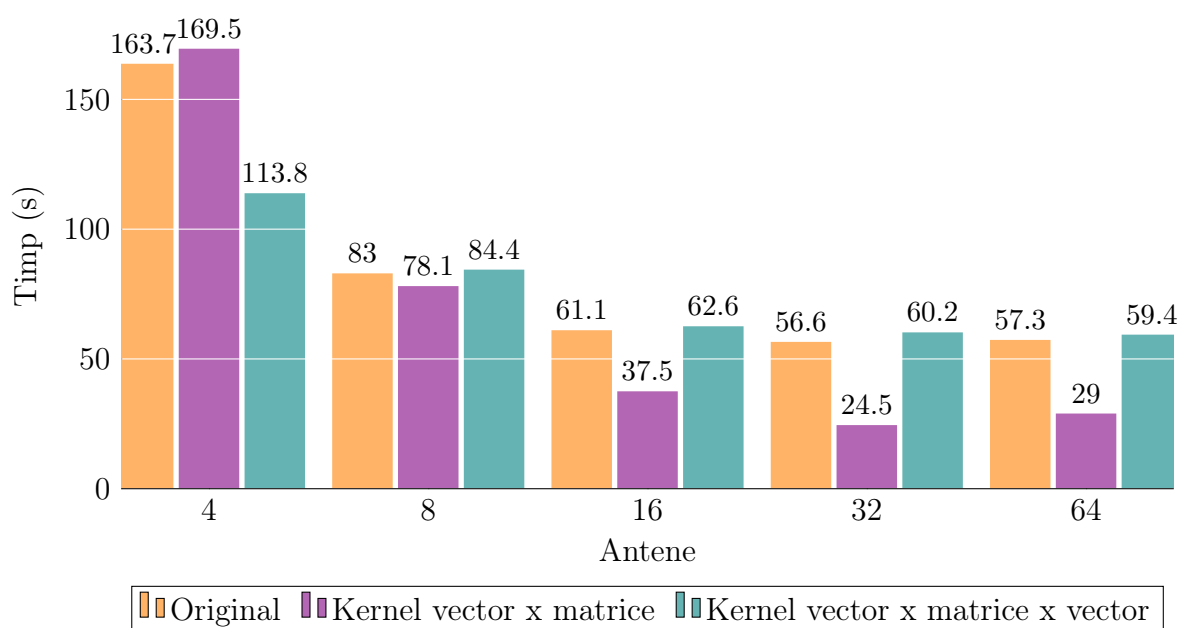


Figura 5.5: Timpul de execuție al lanțului de procesare MUSIC în cele trei variante pentru o dimensiune a spectrului MUSIC de 1024 de elemente

5.3.3 Concluzii

Având în vedere rezultatele obținute, putem considera justificabilă doar folosirea kernelului ce înmulțește un vector linie cu o matrice pătratică în calculul spectrului MUSIC, motiv pentru care, pentru simplitate, ne vom referi în continuare la acesta sub numele de „kernel MUSIC”. Kernelul care efectuează înmulțirea dintre un vector linie, o matrice pătratică și un vector coloană se pretează doar pentru tablouri de dimensiuni mici, până în 4 elemente. Deoarece configurațiile sistemelor radio sunt, cel mai adesea, statice, în sensul că un echipament nu își va schimba în mod dinamic numărul de antene, alegerea unei anumite variante se va face fără niciun fel de *overhead*.

5.4 Evaluarea performanțelor lanțului de procesare MUSIC folosind ConnexArray

În momentul de față, kernelurile create nu se pot folosi simultan în lanțul de procesare MUSIC, deoarece nu există implementată, momentan, o modalitate de gestionare a accesului la resurse pentru procesări diferite pe accelerator. Acest lucru ar presupune asigurarea faptului că execuția unui kernel se face pe un set de date dorit, că un bloc nu citește elementele din coada de reducere destinate altui bloc și, în plus, o metodă de selectare a blocului care primește acces la accelerator. Pentru a stabili dacă un astfel de mecanism este eficient, trebuie efectuată o analiză a gradului de utilizare a acceleratorului relativ la cel al blocului. Dacă se constată că acesta este sub-utilizat, atunci acceleratorul ar putea fi folosit succesiv de mai multe blocuri fără întârzieri prea mari.

O alternativă la implementarea unui astfel de mecanism, în cazul în care se constată că un accelerator nu poate suporta execuțiile mai multor blocuri de procesare, poate fi existența a cel puțin un acceleratoar în plus în FPGA, astfel încât fiecare bloc să aibă suportul său hardware dedicat, cu costul creșterii complexității sistemului și al consumului de putere.

Întrucât niciuna dintre aceste situații nu este, momentan, posibilă, ne vom rezuma la analiza fiecărui kernel integrat separat în lanțul de procesare MUSIC, căruia i-au fost aplicate la intrare patru seturi de date citite din fișiere, fiecare de o dimensiune de 8 MB: două pentru semnale de intrare cosinusoidale cu frecvențe de 10 kHz și 20 kHz și două pentru sursele de zgomot gaussian, iar întregul lanț de procesare a primit următorii parametri:

- Frecvența de eșantionare: 320 kHz
- Numărul de semnale de intrare: 2
- Numărul de antene: 4, 8, 16, 32 sau 64
- Dimensiunea spectrului MUSIC: 1024 elemente
- Dimensiunea capturii autocorelației: 1024 pentru cazul cu 64 de antene și 2048 eșantioane, în rest¹
- Suprapunerea dintre capturi: 256 pentru cazul cu 64 de antene și 512 eșantioane, în rest

¹În cazul sistemelor cu 64 antene, schimbăm dimensiunea capturii autocorelației, deoarece aceasta poate fi maxim 1024 utilizând kernelul pentru autocorelație în care date nu se împart în mai multe blocuri de procesare.

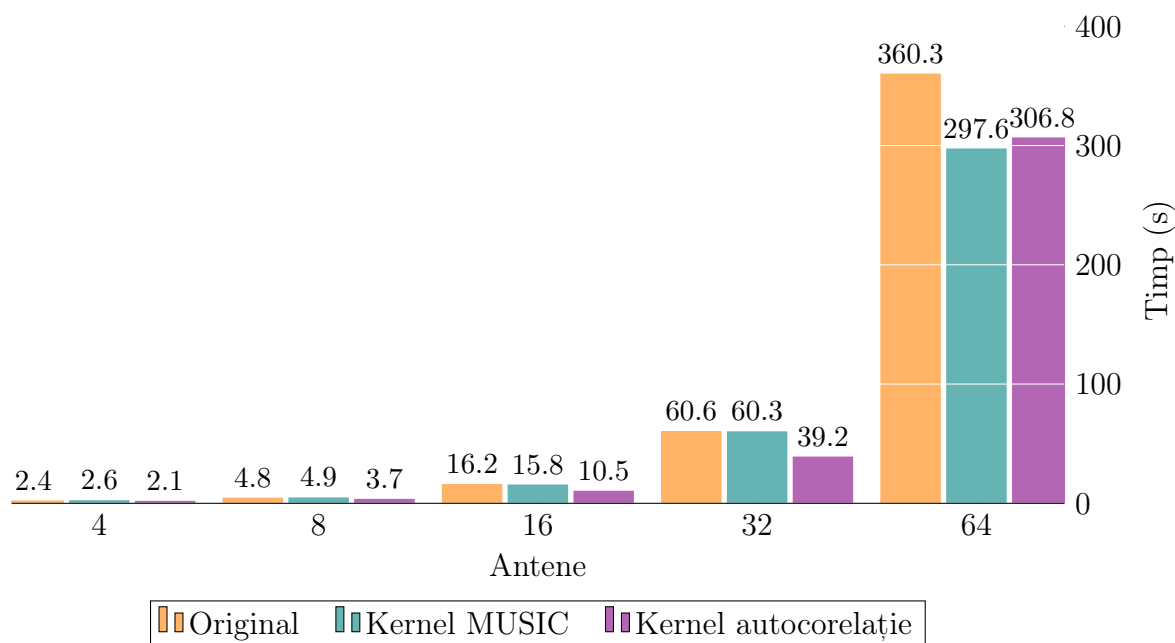


Figura 5.6: Timpul de execuție al lanțului de procesare MUSIC pentru o dimensiune a spectrului MUSIC de 1024 de elemente

Rezultatele obținute pentru timpii de execuție sunt sintetizate în Figura 5.6. Am observat că, per ansamblu, nu există îmbunătățiri semnificative în timpul de execuție al întregului lanț de procesare dacă integrăm, pe rând, câte unul dintre blocurile optimizate, deși evaluate individual fiecare dintre ele a prezentat un timp de execuție mult mai mic decât varianta originală. O îmbunătățire constantă s-a evidențiat doar în privința variantei care folosește kernelul pentru autocorelație, care pentru 16 și 32 de antene a obținut un timp de execuție cu 35% mai redus, iar pentru un număr de 64 de antene ambele variante care folosesc acceleratorul au avut un timp de execuție de aproximativ 5 minute, față de cele 6 minute ale implementării originale.

Motivul pentru care performanțele în întreg ansamblul nu le ating pe cele constatate pentru fiecare bloc în parte provine din faptul că, după s-a putut vedea în Capitolul 3, în Tabelul 3.3, ponderea blocurilor de autocorelație și a celui care calculează spectrul MUSIC în numărul total de cicli de execuție pe procesor a fost similară, de 18,97% și, respectiv, 13,68%, iar blocurile depind unul de celălalt în privința schimbului de date. Optimizându-l doar pe unul dintre ele, celălalt bloc va reprezenta un *bottleneck*, blocând fluxul de date, și timpul de execuție va fi, per total, similar cu cel al variantei originale, lucru observat și cu ajutorul utilitarului **top** disponibil pe sistemul de operare Linux, vizualizând statistici pentru fiecare fir de execuție.

În Tabelele 5.4 și 5.5 sunt prezentate principalele rezultate ale unor profile ale ciclilor de execuție efectuate pe întregul lanț de procesare, în situația în care folosim kernelul pentru autocorelația unui semnal și, respectiv, kernelul care calculează spectrul MUSIC. În Tabelul 5.4 observăm că, atunci când folosim kernelul de autocorelație, sarcina principală de procesare, de până la 43,51% din ciclii totali de execuție, cade în aria blocului care calculează spectrul MUSIC și, invers, când folosim kernelul MUSIC în întregul lanț de procesare, punctul critic ajunge în blocul de autocorelație, care realizează 42,08% din procesare, după cum se observă în Tabelul 5.5, ceea ce dovedește, încă o dată, interdependența dintre cele două blocuri și dificultatea de a le analiza separat.

Overhead	Command	Symbol
43,51%	MUSIC_lin_array	cgemm_
12,16%	autocorrelate_cnx	prepareInData
2,94%	autocorrelate_cnx	std::complex <float>::imag

Tabela 5.4: Profil de execuție pentru lanțul MUSIC folosind kernelul de autocorelație

Overhead	Command	Symbol
42,08%	autocorrelate	cgemm_
3,69%	autocorrelate	arma::eopcore[...]
3,40%	multiply_matrix	--mulsc3

Tabela 5.5: Profil de execuție pentru lanțul MUSIC ce folosește kernelul MUSIC

5.5 Concluzii referitoare la performanțele obținute

În concluzie, performanțele blocurilor considerate individual, constând în timpi de execuție de 4 - 7 ori mai mici, în cazul blocului care folosește kernelul pentru autocorelație, și de 2 ori mai mici, în cazul blocului care folosește kernelul pentru calculul spectrului MUSIC, le fac demne de luat în considerare în implementarea algoritmului MUSIC și, în cazul în care ar putea fi folosite simultan, ar conduce, cu siguranță, și la îmbunătățiri în timpul de execuție al întregului ansamblu, după cum reiese din profilurile de execuție realizate pentru acesta.

O altă îmbunătățire posibilă ar fi existența unui sport hardware pentru conversia datelor din virgulă mobilă în virgulă fixă, operație care poate deveni deosebit de costisitoare, după cum s-a observat în profilurile realizate în Secțiunile 5.2 și 5.3.

S-au observat îmbunătățiri mai accentuate pentru configurații cu mai mult de 16 antene, rezultat relevant, de exemplu, în contextul sistemelor de telecomunicații actuale care folosesc tehnici MIMO sau Massive MIMO cu un număr mare de antene și care pot beneficia de pe urma algoritmilor DoA pentru a obține informații despre sursa unui semnal și pentru a facilita *beamforming*-ul 3D [34].

Având în vedere consumul de putere redus al platformei folosite, se poate lua în considerare folosirea sa în componența unor micro-stații de bază, a căror principală limitare este consumul de putere, sau, dacă resursele de putere sunt mai mari, putem imagina un sistem care să cuprindă o serie de mai multe acceleratoare ConnexArray controlate de către un procesor mai puternic. În rețele celulare eterogene, de exemplu, macro-celulele lucrează alături de micro, pico sau femto celule care au o putere de transmisiune foarte mică și pot fi folosite în mod dinamic în funcție de traficul existent în rețea, pentru a obține o modelare cât mai eficientă a consumului de putere [35].

Posibilitățile de folosire a kernelurilor propuse în algoritmi de localizare a sursei unui semnal nu sunt restrânse doar la algoritmul MUSIC, întrucât și alți algoritmi de localizare (de exemplu, ESPRIT) folosesc autocorelația în procesul de estimare a unghiului de incidență. În plus, generalitatea produselor dintre vectori și matrice le poate face potrivite și pentru alte tipuri de aplicații în procesarea digitală de semnale, iar faptul că kernelurile au fost integrate în blocuri GNU Radio de sine stătătoare facilitează integrarea lor în acestea.

Capitolul 6

Concluzii

Rezultatele obținute în privința performanțelor blocurilor ce utilizează kerneluri ConnexArray demonstrează faptul că acestea pot fi folosite în accelerarea algoritmului de localizare MUSIC pentru un plus de performanță cu un consum de putere redus. Integrarea acestora în blocuri din mediul GNU Radio pentru radio definit software le facilitează utilizarea în aplicații deja existente și le extinde domeniul de aplicabilitate, în contextul unei dezvoltări accentuate a acestui tip de aplicații.

În plus față de implementarea propriu-zisă a blocurilor accelerate folosind ansamblul Connex-ARM, am creat și o metodă generică de testare cu mediul Google Test, care curpinde peste 100 de teste de evaluare a rezoluției și preciziei obținute în diverse configurații ale sistemului, care poate fi utilizată inclusiv cu simulatorul OPINCAA și sistemul hardware.

Evaluarea performanțelor atinse de implementarea algoritmului MUSIC, atât în varianta originală, cât și în cea accelerată, deschide mai multe posibilități de îmbunătățire a acestora. Acest aspect devine important dacă luăm în considerare faptul că, deși algoritmul MUSIC este preferat din motive de precizie, stabilitate și rezoluție, el oferă aceste avantaje cu costul creșterii resurselor computaționale necesare în implementarea sa și este de dorit să se găsească metode prin care acestea pot fi reduse.

Una dintre aceste posibilități este legată de cerința folosirii acceleratorului ConnexArray de mai multe blocuri de procesare, care se poate dovedi extrem de utilă în aplicații radio definite software în care procesarea este distribuită în mai multe blocuri cu sarcini bine definite și vom considera această temă o posibilă direcție de cercetare în viitor.

Contribuții

Contribuțiile principale pe care le-am adus în realizarea acestui proiect sunt:

- Analiza unei implementări generale a algoritmului MUSIC și stabilirea punctelor critice în performanța sa.
- Crearea și implementarea unor algoritmi pentru kerneluri ConnexArray corespunzătoare punctelor critice identificate.
- Integrarea kernelurilor create în blocuri de sine stătătoare în mediul GNU Radio, care pot fi incluse în lanțul de procesare MUSIC.
- Testarea lanțului de procesare în condiții de simulare și evaluarea performanțelor blocurilor create și a întregului ansamblu în raport cu implementarea originală.
- Pe parcursul proiectului, am identificat și contribuit la rezolvarea unor probleme ale utilităților folosite:
 - Identificarea și rezolvarea unei erori în realizarea reducățiilor pe anumite blocuri de elemente selectate în simulatorul OPINCAA.
 - Identificarea unei erori în implementarea hardware a acceleratorului ConnexArray legată de procesarea reducățiilor pentru numere cu semn.
 - Identificarea și soluționarea unei erori în implementarea algoritmului MUSIC folosită în realizarea medierii antegrade-retrograde a estimatului autocorelației.
 - Identificarea unei erori în simulatorul OPINCAA atunci când un kernel trebuie să livreze un număr prea mare de reducății.

Anexa 1

Testarea lanțului de procesare MUSIC

1.1 Declararea claselor necesare în testare

```
1 #ifndef DOA_TEST_H_
2 #define DOA_TEST_H_
3
4 #include <array>
5 #include <string>
6 #include <vector>
7
8 #include <gnuradio/analog/sig_source_waveform.h>
9 #include <gnuradio/analog/noise_type.h>
10
11 // Used to access the command line arguments from inside the test cases
12 extern int my_argc;
13 extern char **my_argv;
14
15 template <int SIGNAL_SOURCES>
16 class flowgraph_parameters
17 {
18     template <typename T>
19     using array = std::array<T, SIGNAL_SOURCES>;
20
21 public:
22     double sample_rate;
23     float norm_spacing;
24     int num_array_elements;
25     int p_spectrum_length;
26     int snapshot_size;
27     int overlap_size;
28     int nr_output_items;
29     array<double> freq;
30     array<int> theta_deg;
31     array<double> signal_amplitude;
32     array<double> noise_amplitude;
33     array<gr::analog::gr_waveform_t> waveform;
34     array<gr::analog::noise_type_t> noise_type;
35     std::string distributionFIFO;
36     std::string reductionFIFO;
37     std::string writeFIFO;
38     std::string readFIFO;
39
40     flowgraph_parameters() {}
41
42     flowgraph_parameters(flowgraph_parameters const &fg_param) :
43         sample_rate(fg_param.sample_rate),
44         norm_spacing(fg_param.norm_spacing),
45         num_array_elements(fg_param.num_array_elements),
46         p_spectrum_length(fg_param.p_spectrum_length),
47         snapshot_size(fg_param.snapshot_size),
48         overlap_size(fg_param.overlap_size),
49         nr_output_items(fg_param.nr_output_items),
50         freq(fg_param.freq),
51         theta_deg(fg_param.theta_deg),
52         signal_amplitude(fg_param.signal_amplitude),
53         noise_amplitude(fg_param.noise_amplitude),
54         waveform(fg_param.waveform),
```

```

55         noise_type(fg_param.noise_type),
56         distributionFIFO(fg_param.distributionFIFO),
57         reductionFIFO(fg_param.reductionFIFO),
58         writeFIFO(fg_param.writeFIFO),
59         readFIFO(fg_param.readFIFO)
60     {}
61
62     flowgraph_parameters &set_sample_rate(double sample_rate_) {
63         sample_rate = sample_rate_;
64         return *this;
65     }
66
67     flowgraph_parameters &set_norm_spacing(float norm_spacing_) {
68         norm_spacing = norm_spacing_;
69         return *this;
70     }
71
72     flowgraph_parameters &set_num_array_elements(int num_array_elements_) {
73         num_array_elements = num_array_elements_;
74         return *this;
75     }
76
77     flowgraph_parameters &set_p_spectrum_length(int p_spectrum_length_) {
78         p_spectrum_length = p_spectrum_length_;
79         return *this;
80     }
81
82     flowgraph_parameters &set_snapshot_size(int snapshot_size_) {
83         snapshot_size = snapshot_size_;
84         return *this;
85     }
86
87     flowgraph_parameters &set_overlap_size(int overlap_size_) {
88         overlap_size = overlap_size_;
89         return *this;
90     }
91
92     flowgraph_parameters &set_nr_output_items(int nr_output_items_) {
93         nr_output_items = nr_output_items_;
94         return *this;
95     }
96
97     flowgraph_parameters &set_freq(const array<double> &freq_) {
98         freq = freq_;
99         return *this;
100    }
101
102     flowgraph_parameters
103     &set_theta_deg(const array<int> &theta_deg_) {
104         theta_deg = theta_deg_;
105         return *this;
106     }
107
108     flowgraph_parameters &set_signal_amplitude(
109         const array<double> &signal_amplitude_) {
110         signal_amplitude = signal_amplitude_;
111         return *this;
112     }
113
114     flowgraph_parameters &set_noise_amplitude(
115         const array<double> &noise_amplitude_) {
116         noise_amplitude = noise_amplitude_;
117         return *this;
118     }
119
120     flowgraph_parameters &set_waveform(
121         const array<gr::analog::gr_waveform_t> &waveform_) {
122         waveform = waveform_;
123         return *this;
124     }
125
126     flowgraph_parameters &set_noise_type(
127         const array<gr::analog::noise_type_t> &noise_type_) {
128         noise_type = noise_type_;
129         return *this;
130     }
131
132     flowgraph_parameters
133     &set_distributionFIFO(std::string distributionFIFO_) {
134         distributionFIFO = distributionFIFO_;

```



```

135         return *this;
136     }
137
138     flowgraph_parameters
139     &set_reductionFIFO(std::string reductionFIFO_) {
140         reductionFIFO = reductionFIFO_;
141         return *this;
142     }
143
144     flowgraph_parameters
145     &set_writeFIFO(std::string writeFIFO_) {
146         writeFIFO = writeFIFO_;
147         return *this;
148     }
149
150     flowgraph_parameters
151     &set_readFIFO(std::string readFIFO_) {
152         readFIFO = readFIFO_;
153         return *this;
154     }
155 };
156
157 template <int SIGNAL_SOURCES>
158 class doa_flowgraph
159 {
160 public:
161     flowgraph_parameters<SIGNAL_SOURCES> fg_param;
162
163     doa_flowgraph(const flowgraph_parameters<SIGNAL_SOURCES> &fg_param_) :
164         fg_param(fg_param_)
165     {}
166
167     std::vector<float> build_flowgraph();
168 };
169 #endif // DOA_TEST_H_

```

Listarea 1.1: Declararea claselor necesare în testare

1.2 Definirea metodei de construcție a unui lanț de procesare

```

1 #include "doa_test.h"
2
3 #include <gnuradio/top_block.h>
4 #include <gnuradio/analog/sig_source_c.h>
5 #include <gnuradio/analog/noise_source_c.h>
6 #include <gnuradio/blocks/add_cc.h>
7 #include <gnuradio/blocks/throttle.h>
8 #include <gnuradio/blocks/multiply_matrix_cc.h>
9 #include <gnuradio/blocks/null_sink.h>
10 #include <gnuradio/blocks/vector_to_streams.h>
11 #include <gnuradio/blocks/head.h>
12 #include <gnuradio/blocks/vector_sink_f.h>
13 #include <gnuradio/gr_complex.h>
14
15 #include <doa/autocorrelateConnexKernel.h>
16 #include <doa/autocorrelate.h>
17 #include <doa/MUSIC_lin_array.h>
18 #include <doa/MUSIC_lin_array_cnx.h>
19 #include <doa/find_local_max.h>
20
21 #include <boost/math/constants/constants.hpp> // for type float PI
22
23 #include <cmath>
24 #include <complex>
25 #include <chrono>
26 #include <thread>
27
28 using namespace std::complex_literals;
29
30 template class doa_flowgraph<2>;
31
32 template <int SIGNAL_SOURCES>
33 std::vector<float> doa_flowgraph<SIGNAL_SOURCES>::build_flowgraph()

```

```

34 {
35     gr::top_block_sptr tb = gr::make_top_block("run_MUSIC");
36
37     const float pi = boost::math::constants::pi<float>();
38     const int num_targets = 2;
39
40     float theta0 = pi * fg_param.theta_deg[0] / 180;
41     float theta1 = pi * fg_param.theta_deg[1] / 180;
42
43     std::vector<gr_complex> amv0(fg_param.num_array_elements);
44     std::vector<gr_complex> amv1(fg_param.num_array_elements);
45     std::vector<std::vector<gr_complex>> array_manifold_matrix(
46         fg_param.num_array_elements,
47         std::vector<gr_complex>(num_targets)
48     );
49
50     float ant_loc_val;
51
52     if (fg_param.num_array_elements % 2 == 1) {
53         ant_loc_val = (float)fg_param.num_array_elements / 2;
54     } else {
55         ant_loc_val = (float)fg_param.num_array_elements / 2 - 0.5;
56     }
57
58     // can't multiply by an integer (2) when we have -lif so it's a workaround
59     float temp_norm_spacing = fg_param.norm_spacing * 2;
60     for (int i = 0; i < fg_param.num_array_elements; i++) {
61         amv0[i] = std::exp(-lif * temp_norm_spacing * pi * ant_loc_val *
62             cosf(theta0));
63         amv1[i] = std::exp(-lif * temp_norm_spacing * pi * ant_loc_val *
64             cosf(theta1));
65         array_manifold_matrix[i][0] = amv0[i];
66         array_manifold_matrix[i][1] = amv1[i];
67
68         ant_loc_val--;
69     }
70
71     /*=====
72     * Blocks
73     *=====*/
74
75     gr::analog::sig_source_c::sptr sig_src0 = gr::analog::sig_source_c::make(
76         fg_param.sample_rate,           // sample rate
77         fg_param.waveform[0],           // waveform used
78         fg_param.freq[0],               // wave frequency
79         fg_param.signal_amplitude[0],   // amplitude
80         0                                // offset
81     );
82
83     gr::analog::sig_source_c::sptr sig_src1 = gr::analog::sig_source_c::make(
84         fg_param.sample_rate,           // sample rate
85         fg_param.waveform[1],           // waveform used
86         fg_param.freq[1],               // wave frequency
87         fg_param.signal_amplitude[1],   // amplitude
88         0                                // offset
89     );
90
91     gr::analog::noise_source_c::sptr n_src0 =
92     gr::analog::noise_source_c::make(
93         fg_param.noise_type[0],         // type of noise
94         fg_param.noise_amplitude[0]     // amplitude of the noise
95     );
96
97     gr::analog::noise_source_c::sptr n_src1 =
98     gr::analog::noise_source_c::make(
99         fg_param.noise_type[1],         // type of noise
100        fg_param.noise_amplitude[1]     // amplitude of the noise
101     );
102
103     gr::blocks::throttle::sptr thr0 = gr::blocks::throttle::make(
104         sizeof(gr_complex),             // item size
105         fg_param.sample_rate            // sample rate
106     );
107
108     gr::blocks::add_cc::sptr add0 = gr::blocks::add_cc::make();
109
110     gr::blocks::add_cc::sptr add1 = gr::blocks::add_cc::make();
111
112     gr::blocks::multiply_matrix_cc::sptr mult_by_matrix0 =
113     gr::blocks::multiply_matrix_cc::make(

```

```

114         array_manifold_matrix,
115         gr::block::TPP_ALL_TO_ALL
116     );
117
118     gr::doa::autocorrelate::sptr autocorr0 =
119         gr::doa::autocorrelate::make(
120             fg_param.num_array_elements,
121             fg_param.snapshot_size,
122             fg_param.overlap_size,
123             1
124         );
125
126     gr::doa::MUSIC_lin_array::sptr music_lin_array0 =
127         gr::doa::MUSIC_lin_array::make(
128             fg_param.norm_spacing,
129             num_targets,
130             fg_param.num_array_elements,
131             fg_param.p_spectrum_length
132         );
133
134     gr::doa::find_local_max::sptr find_local_max0 =
135         gr::doa::find_local_max::make(
136             num_targets,           // maximum number of local maximums
137             fg_param.p_spectrum_length,
138             0.0,                   // min value of index vector
139             180.0                  // max value of index vector
140         );
141
142     // Null sink for locations
143     gr::blocks::null_sink::sptr null_sink0 =
144         gr::blocks::null_sink::make(
145             sizeof(float) * num_targets // item size
146         );
147
148     gr::blocks::vector_to_streams::sptr v_to_s0 =
149         gr::blocks::vector_to_streams::make(
150             sizeof(float),         // item size
151             num_targets             // number of streams
152         );
153
154     // Limits the number of output items
155     gr::blocks::head::sptr head0 = gr::blocks::head::make(
156         sizeof(float),           // size of stream item
157         fg_param.nr_output_items // number of output items
158     );
159     gr::blocks::head::sptr head1 = gr::blocks::head::make(
160         sizeof(float),           // size of stream item
161         fg_param.nr_output_items // number of output items
162     );
163
164     gr::blocks::vector_sink_f::sptr v_sink0 = gr::blocks::vector_sink_f::make();
165     gr::blocks::vector_sink_f::sptr v_sink1 = gr::blocks::vector_sink_f::make();
166
167     /*=====
168     * Connections
169     *=====*/
170
171     tb->connect(sig_src0, 0, add0, 0);
172     tb->connect(sig_src1, 0, add1, 0);
173
174     tb->connect(n_src0, 0, add0, 1);
175     tb->connect(n_src1, 0, thr0, 0);
176     tb->connect(thr0, 0, add1, 1);
177
178     tb->connect(add0, 0, mult_by_matrix0, 0);
179     tb->connect(add1, 0, mult_by_matrix0, 1);
180
181     for (int i = 0; i < fg_param.num_array_elements; i++) {
182         tb->connect(mult_by_matrix0, i, autocorr0, i);
183     }
184
185     tb->connect(autocorr0, 0, music_lin_array0, 0);
186     tb->connect(music_lin_array0, 0, find_local_max0, 0);
187     tb->connect(find_local_max0, 0, null_sink0, 0);
188
189     tb->connect(find_local_max0, 1, v_to_s0, 0);
190     tb->connect(v_to_s0, 0, head0, 0);
191     tb->connect(v_to_s0, 1, head1, 0);
192
193     tb->connect(head0, 0, v_sink0, 0);

```

```

194     tb->connect(head1, 0, v_sink1, 0);
195
196     /*=====
197     * Run the graph
198     *=====*/
199     tb->start();
200
201     std::thread t([&tb]{
202         // Let run for 5 seconds
203         std::this_thread::sleep_for(std::chrono::seconds(5));
204         tb->stop();
205         tb->wait();
206     });
207     t.join();
208
209     std::vector<float> out_data0 = v_sink0->data();
210     std::vector<float> out_data1 = v_sink1->data();
211
212     if ((out_data0.size() > 0) && (out_data1.size() > 0))
213     {
214         size_t last_idx0 = out_data0.size() - 1;
215         size_t last_idx1 = out_data1.size() - 1;
216
217         std::cout << "Processed samples: " << out_data0.size() << std::endl;
218
219         std::vector<float> out_data;
220         out_data.push_back(out_data0[last_idx0]);
221         out_data.push_back(out_data1[last_idx1]);
222         return out_data;
223     } else {
224         return {-1, -1};
225     }
226 }

```

Listarea 1.2: Definirea metodei de construcție a unui lanț de procesare

Anexa 2

Kerneluri ConnexArray

2.1 Kernel pentru înmulțirea a doi vectori

```
1 void init_kernel(void) {
2     BEGIN_KERNEL("init_kernel");
3     EXECUTE_IN_ALL(
4         R25 = 0; // locatia din memoria locala de unde se incarca primul vector
5         R26 = 511; // locatia din memoria locala de unde se incarca al 2-lea vector
6         R30 = 1; // numarul 1; util in calcule
7         R31 = 0; // numarul 0; util in calcule
8     )
9 }
10
11 void multiply_kernel(void) {
12     BEGIN_KERNEL("multiply_arr_mat");
13     EXECUTE_IN_ALL(
14         R1 = LS[R25]; // Pasul 1 - partea 1
15         R2 = LS[R26]; // Pasul 1 - partea a 2-a
16         R29 = INDEX; // Incarcarea indexului unui PE
17
18         R3 = R1 * R2; // Pasul 2
19         R3 = MULT_HIGH();
20
21         CELL_SHL(R2, R30); // Pasul 5
22         NOP;
23         R4 = SHIFT_REG;
24         R4 = R1 * R4; // Pasul 7
25         R4 = MULT_HIGH();
26
27         CELL_SHR(R2, R30); // Pasul 6
28         NOP;
29         R5 = SHIFT_REG;
30         R5 = R1 * R5; // Pasul 8
31         R5 = MULT_HIGH();
32
33         R9 = INDEX; // Pasul 9 - selectia PE cu index impar
34         R9 = R9 & R30;
35         R7 = (R9 == R30);
36     )
37
38     EXECUTE_WHERE_EQ( // Instructiunile se executa doar in PE impar
39         R3 = R31 - R3; // Pasul 3
40
41         R4 = R5; // Pasul 9 - mutarea R5 -> R4
42     )
43
44     EXECUTE_IN_ALL(
45         REDUCE(R3); // Pasul 3
46         REDUCE(R4); // Pasul 10
47     )
48     END_KERNEL("multiply_arr_mat");
49 }
```

Listarea 2.1: Kernel de inițializare a conținutului registrelor

2.2 Kernel pentru înmulțirea mai multor vectori cu aceeași matrice

```

1 void MUSIC_lin_array_cnx_impl::init_kernel(int size_of_block)
2 {
3     BEGIN_KERNEL("initKernel");
4     EXECUTE_IN_ALL(
5         R10 = 0;
6         R11 = 1;
7         R12 = 0;
8         R13 = 900;
9         R14 = size_of_block;          // Equal to ARR_SIZE_C; dimension of the blocks
10                                     // on which reduction is performed at once
11         R8 = INDEX;                  // Select only the odd PEs
12     )
13     END_KERNEL("initKernel");
14 }
15
16 void MUSIC_lin_array_cnx_impl::init_index(void)
17 {
18     BEGIN_KERNEL("initIndex");
19     EXECUTE_IN_ALL(
20         R2 = LS[R13];                // load input matrix
21         REDUCE(R0);
22     )
23     END_KERNEL("initIndex");
24 }
25
26 void MUSIC_lin_array_cnx_impl::multiply_kernel(
27     int LS_per_iteration, int size_reduction_block, int blocks_to_reduce)
28 {
29     BEGIN_KERNEL("multiplyArrMatKernel");
30     EXECUTE_IN_ALL(
31         R12 = 0;                      // reset array LS index
32     )
33
34     for (int i = 0; i < LS_per_iteration; i++) {
35         EXECUTE_IN_ALL(
36             R1 = LS[R12];              // load input array
37             R0 = INDEX;                // Used later to select PEs for reduction
38             R6 = size_reduction_block; // Used to select blocks for reduction
39
40             R3 = R1 * R2;               // a1 * a2, b1 * b2
41             R3 = MULT_HIGH();
42
43             CELL_SHL(R2, R11);          // Bring b2 to the left to calc b2 * a1
44             NOP;
45             R4 = SHIFT_REG;
46             R4 = R1 * R4;               // a1 * b2
47             R4 = MULT_HIGH();
48
49             CELL_SHR(R2, R11);
50             NOP;
51             R5 = SHIFT_REG;
52             R5 = R1 * R5;               // b1 * a2
53             R5 = MULT_HIGH();
54
55             R9 = R8 & R11;
56             R7 = (R9 == R11);
57             NOP;
58         )
59
60         EXECUTE_WHERE_EQ(              // Only in the odd PEs
61             // Getting -b1 * b2 in each odd cell
62             R3 = R10 - R3;              // All partial real parts are in R3
63
64             R4 = R5;                    // All partial imaginary parts are now in R4
65         )
66
67         REPEAT_X_TIMES(blocks_to_reduce);
68         EXECUTE_IN_ALL(
69             R7 = (R0 < R6);              // Select only blocks of PEs at a time
70             NOP;
71         )
72         EXECUTE_WHERE_LT(
73             R0 = 129;                    // A random number > 128 so these PEs won't
74                                     // be selected again
75             REDUCE(R3);                  // Real part

```

```

76         REDUCE(R4);                // Imaginary part
77     )
78     EXECUTE_IN_ALL(
79         R6 = R6 + R14;              // Go to the next block of PEs
80     )
81     END_REPEAT;
82
83     EXECUTE_IN_ALL(
84         R12 = R12 + R11;            // Go to the next LS
85     )
86 }
87 END_KERNEL("multiplyArrMatKernel");
88 }

```

Listarea 2.2: Kernel pentru înmulțirea unui vector linie cu o matrice pătratică,

2.3 Kernel pentru înmulțirea înlănțuită dintre un vector linie, o matrice și un vector coloană

```

1 void MUSIC_lin_array_cnx_impl::init_chained(int size_red_block, int LS_mat_)
2 {
3     BEGIN_KERNEL("initKernelChained");
4     EXECUTE_IN_ALL(
5         R26 = LS_mat_;              // From where to start reading the matrix
6         R30 = 1;
7         R31 = 0;
8         R28 = size_red_block;       // dimension of the blocks on which reduction is
9                                     // performed at once
10        R9 = INDEX;                 // Select only the odd PEs
11    )
12    END_KERNEL("initKernelChained");
13 }
14
15 void MUSIC_lin_array_cnx_impl::init_index_chained(void)
16 {
17     BEGIN_KERNEL("initIndexChained");
18     EXECUTE_IN_ALL(
19         R2 = LS[R26];               // Load new matrix
20     )
21     END_KERNEL("initIndexChained");
22 }
23
24 void MUSIC_lin_array_cnx_impl::multiply_chained(int LS_per_execution,
25 int size_red_block, int nr_red_blocks, int LS_in_arr_, int LS_final_real_,
26 int LS_final_imag_)
27 {
28     BEGIN_KERNEL("multChained");
29     EXECUTE_IN_ALL(
30         R25 = LS_in_arr_;           // Reset the array index
31         R24 = LS_final_real_;       // Reset the real part of the final array index
32         R23 = LS_final_imag_;       // Reset the imag part of the final array index
33     )
34
35     for (int i = 0; i < LS_per_execution; i++) {
36         EXECUTE_IN_ALL(
37             R1 = LS[R25];            // Load input array
38             R15 = LS[R24];           // Load real part of the final array
39             R16 = LS[R23];           // Load imag part of the final array
40             R29 = INDEX;             // Used later to select PEs for reduction
41             R27 = size_red_block;    // Used to select blocks for reduction
42
43             R16 = R31 - R16;         // Invert imag part of the final array
44
45             R3 = R1 * R2;             // a1 * a2, b1 * b2
46             R3 = MULT_HIGH();        // Get partial real parts of the first arr x
47                                     // mat product
48             CELL_SHL(R1, R30);        // Bring b1 to the left to calc b1 * a2
49             NOP;
50             R4 = SHIFT_REG;
51             R4 = R4 * R2;             // b1 * a2
52             R4 = MULT_HIGH();
53
54             CELL_SHR(R1, R30);
55             NOP;

```

```

56     R5 = SHIFT_REG;
57     R5 = R5 * R2;           // b2 * a1
58     R5 = MULT_HIGH();
59
60     R10 = R9 & R30;
61     R7 = (R10 == R30);
62     NOP;
63 )
64
65 EXECUTE_WHERE_EQ(           // Only in the odd PEs
66     R3 = R31 - R3;           // Invert sign of these partial real parts
67
68     R4 = R5;                 // All partial imaginary parts of the first
69                               // product are now in R4
70 )
71
72 EXECUTE_IN_ALL(
73     R3 = R3 * R15;           // Multiply by the real part of the elements
74     R3 = MULT_HIGH();        // in the final array
75
76     R4 = R4 * R16;           // Multiply the partial imag parts of the
77                               // first product by the imag part of the final array
78     R4 = MULT_HIGH();
79 )
80
81 // We reduce blocks of size mat_size_c; Half of the final result is in
82 // R3 and the other half is in R4
83 REPEAT_X_TIMES(nr_red_blocks);
84     EXECUTE_IN_ALL(
85         R7 = (R29 < R27);     // Select only blocks of PEs at a time
86         NOP;
87     )
88     EXECUTE_WHERE_LT(
89         R29 = 129;            // A random number > 128 so these PEs won't be
90                               // selected again
91         REDUCE(R3);           // Real part
92         REDUCE(R4);           // Imaginary part
93     )
94     EXECUTE_IN_ALL(
95         R27 = R27 + R28;      // Go to the next block of PEs
96     )
97 END_REPEAT;
98
99 EXECUTE_IN_ALL(
100     R25 = R25 + R30;          // Go to the next LS
101     R24 = R24 + R30;
102     R23 = R23 + R30;
103 )
104 }
105 END_KERNEL("multChained");
106 }

```

Listarea 2.3: Kernel pentru înmulțirea înlănțuită dintre un vector linie, o matrice și un vector coloană în situația în care cel puțin o matrice poate fi scrisă pe o linie din LS

2.4 Kernel pentru realizarea autocorelației

```

1  void autocorrelate_cnx_impl::initKernel(const int &LS_per_row_) {
2      BEGIN_KERNEL("initKernel");
3      EXECUTE_IN_ALL(
4          R0 = LS_per_row_;
5          R29 = 1;
6          R28 = 0;
7          R25 = 2;
8      )
9      END_KERNEL("initKernel");
10 }
11
12 void autocorrelate_cnx_impl::autocorrelationKernel(
13     const int &iterations_per_array)
14 {
15     BEGIN_KERNEL("autocorrelationKernel");
16     for (int i = 0; i < n_rows; i++) {
17         for (int j = i; j < n_rows; j++) {

```



```

18 EXECUTE_IN_ALL(
19     R14 = i;
20     R15 = j;
21
22     R30 = R14 * R0;           // go to line i
23     R31 = R15 * R0;           // go to col j
24 )
25
26 REPEAT_X_TIMES(iterations_per_array);
27 EXECUTE_IN_ALL(
28     R1 = LS[R30];
29     R2 = LS[R31];
30     R3 = R1 * R2;
31     R3 = MULT_HIGH();         // re1 * re2, im1 * im2
32
33     CELL_SHL(R2, R29);
34     NOP;
35     R4 = SHIFT_REG;
36     R4 = R4 * R1;
37     R4 = MULT_HIGH();         // re1 * im2
38     R4 = R28 - R4;             // The column is conjugated => negate these
39
40     CELL_SHR(R2, R29);
41     NOP;
42     R5 = SHIFT_REG;
43     R5 = R5 * R1;
44     R5 = MULT_HIGH();         // re2 * im1;
45
46     R6 = INDEX;               // Select only the odd PEs
47     R6 = R6 & R29;
48     R7 = (R6 == R29);
49     NOP;
50 )
51
52 EXECUTE_WHERE_EQ(
53     R4 = R5;                   // All partial imaginary parts are now in R4
54 )
55
56 EXECUTE_IN_ALL(
57     REDUCE(R3);
58     REDUCE(R4);
59
60     R30 = R30 + R29;           // Go to next chunk on line & col
61     R31 = R31 + R29;
62 )
63 END_REPEAT;
64 }
65 }
66 END_KERNEL("autocorrelationKernel");
67 }

```

Listarea 2.4: Kernel pentru calculul autocorelației unui semnal

Anexa 3

Evaluarea performanțelor kernelurilor pentru acceleratorul ConnexArray

3.1 Măsurarea timpului de execuție al unui program

```
1 auto start = std::chrono::system_clock::now();  
2  
3 tb->start();  
4 tb->wait();  
5  
6 auto end = std::chrono::system_clock::now();  
7 auto elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(end - start);  
8 std::cout << "Wall time: " << elapsed.count() << std::endl;
```

Listarea 3.1: Măsurarea timpului de execuție al unui program

Bibliografie

- [1] Bira C., Gugu L., Hobincu R., Petrica L., Codreanu V., and Cotofana S. *An Energy Effective SIMD Accelerator for Visual Pattern Matching*. 4th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies, 2013.
- [2] Santarini M. *Zynq-7000 EPP sets stage for new era of innovations*. Xcell Journal, 75:8–13, 2011.
- [3] Ralph Schmidt. Multiple emitter location and signal parameter estimation. *IEEE transactions on antennas and propagation*, 34(3):276–280, 1986.
- [4] Ettus Research. Phase synchronization capability of TwinRX Daughterboards and DoA estimation. <https://github.com/EttusResearch/gr-doa/wiki>. Accesat pe 11-06-2017.
- [5] GNU Radio Framework. <https://www.gnuradio.org/>. Accesat pe 11-06-2017.
- [6] C. Bira, R. Hobincu, L. Petrica, V. Codreanu, and S. Cotofana. *Energy-Efficient Computation of L1 and L2 Normson a FPGA SIMD Accelerator, with Applications to Visual Search*. Advances in Information Science and Applications - Volume II.
- [7] B. Aparna V. Krishnaveni, T. Kesavamurthy. *Beamforming for Direction-of-Arrival (DOA) Estimation-A Survey*. International Journal of Computer Applications (0975 – 8887) Volume 61– No.11, Ianuarie 2013.
- [8] Dhusar Kumar Mondal. *Studies of Different Direction of Arrival (DOA) Estimation Algorithm for Smart Antenna in Wireless Communication*. IJECT Vol. 4, Issue SPL - 1, Iunie - Martie 2013.
- [9] Panayiotis I. Ioannides Constantine A. Balanis. *Introduction to Smart Antennas*. Morgan & Claypool Publishers, 2007.
- [10] Rias Muhamed. *Direction of arrival estimation using antenna arrays*. PhD thesis, Virginia Tech, 1996.
- [11] Gene H Golub and Charles F Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.
- [12] Firasta N., Buxton M., Jinbo P., Nasri K., and Kuo S. *Intel avx: New frontiers in performance improvements and energy efficiency*. Intel white paper, 2008.
- [13] Procesorul ARM Cortex-A9. <https://www.arm.com/products/processors/cortex-a/cortex-a9.php>. Accesat pe 11-06-2017.
- [14] David Kirk et al. Nvidia cuda software and gpu parallel computing architecture. In *ISMM*, volume 7, pages 103–104, 2007.
- [15] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–73, 2010.

-
- [16] Gheorghe Stefan. The ca1024: A massively parallel processor for cost-effective hdtv. In *Spring Processor Forum: Power-Efficient Design*, pages 15–17, 2006.
 - [17] Călin Bîră, Lucian Petrică, and Radu Hobincu. Opincaa: A light-weight and flexible programming environment for parallel simd accelerators. *Science and Technology*, 16(4):336–350, 2013.
 - [18] SDR Forum. *SDRF Cognitive Radio Definitions*. SDRF-06-R-0011-V1.0.0, Noiembrie 2007.
 - [19] Joseph Mitola. Software radios: Survey, critical evaluation and future directions. *IEEE Aerospace and Electronic Systems Magazine*, 8(4):25–36, 1993.
 - [20] Hüseyin Arslan. *Cognitive radio, software defined radio, and adaptive wireless systems*, volume 10. Springer, 2007.
 - [21] Scheduler GNU Radio. <http://www.trondeau.com/blog/2013/9/15/explaining-the-gnu-radio-scheduler.html>. Accesat pe 11-06-2017.
 - [22] Ousmane Abdoulaye Oumar, Ming Fei Siyau, and Tariq P Sattar. Comparison between music and esprit direction of arrival estimation algorithms for wireless communication systems. In *Future Generation Communication Technology (FGCT), 2012 International Conference on*, pages 99–103. IEEE, 2012.
 - [23] Harry L Van Trees. *Optimum array processing: Part IV of detection, estimation and modulation theory*, volume 1. Wiley Online Library, 2002.
 - [24] Conrad Sanderson and Ryan Curtin. *Armadillo: a template-based C++ library for linear algebra*. Journal of Open Source Software, Vol. 1, pp. 26, 2016.
 - [25] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.
 - [26] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.
 - [27] David M Beazley. Tcl and SWIG as a C/C++ development tool. *Department of Computer Science, University of Chicago*, 1998.
 - [28] Vincent M Weaver. Linux perf_event features and overhead. In *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*, volume 13, 2013.
 - [29] Perf tools documentation. <https://perf.wiki.kernel.org/index.php/Tutorial>. Accesat pe 11-06-2017.
 - [30] Brendan Gregg. The flame graph. *Communications of the ACM*, 59(6):48–57, 2016.
 - [31] Google test framework. <https://github.com/google/googletest/blob/master/googletest/docs/Primer.md>. Accesat pe 11-06-2017.
 - [32] Biblioteca OpenBLAS. <http://www.openblas.net/>. Accesat pe 11-06-2017.
 - [33] Repository openblas. https://github.com/xianyi/OpenBLAS/tree/develop/kernel/x86_64. Accesat pe 11-06-2017.

-
- [34] Ting Wang, Bo Ai, Ruisi He, and Zhangdui Zhong. Two-dimension direction-of-arrival estimation for massive mimo systems. *IEEE Access*, 3:2122–2128, 2015.
 - [35] Josip Lorincz, Tonko Garma, and Goran Petrovic. Measurements and modelling of base station power consumption under real traffic loads. *Sensors*, 12(4):4281–4310, 2012.