UNIVERSITATEA NAȚIONALĂ DE ȘTIINȚĂ ȘI TEHNOLOGIE POLITEHNICA BUCUREȘTI

Școala Doctorală de Electronică, Telecomunicații și Tehnologia Informației

Decizie nr. 126 din 09-11-2023

REZUMAT TEZĂ DE DOCTORAT

Ing. Mihai ANTONESCU

Optimizări algoritmice și arhitecturale pentru un accelerator hardware de tip Map-Reduce

Algorithmical and Architectural Improvements for a Map-Reduce Hardware Accelerator

---

COMISIA DE DOCTORAT

Prof. Dr. Ing. Gheorghe Brezeanu Universitatea Națională de Știință și Tehnologie Politehnica București	Președinte
Prof. Dr. Ing. Gheorghe Ștefan Universitatea Națională de Știință și Tehnologie Politehnica București	Conducător de doctorat
Prof. Dr. Ing. Corneliu Burileanu Universitatea Națională de Știință și Tehnologie Politehnica București	Referent
Prof. Dr. Ing. Aurel-Ștefan Gontean Universitatea Politehnica din Timișoara	Referent
Prof. Dr. Ing. Dan Nicula Universitatea Transilvania Brașov	Referent

BUCUREȘTI 2024

Cuprins

Capitolul 1
Introducere

1.1 Prezentarea domeniului calculului paralel

Calculul paralel reprezintă astăzi majoritatea computației ce se efectuează. Mașinile de tip mono-procesor au fost puse pe o poziție secundară, fiind folosite pentru calcul embedded sau în situații unde puterea de calcul este mai puțin importantă decât alți factori. În cadrul acestui domeniu, în ultimele decenii, se poate observa o schimbare de paradigmă și de abordare. Majoritatea soluțiilor de calcul paralel folosite pentru computație intensă cad în una din doua categorii: mașini existente deja pentru alt domeniu și adaptate pentru aplicația curentă și circuite specifice doar aplicației curente (implementate ca ASIC sau FPGA). Aceste mașini sunt apoi conectate la un calculator gazdă pentru a forma un sistem de calcul eterogen.

Nu exista o soluție bună în toate situațiile, fiecare fiind în mod unic afectată de combinația de memorie (din punct de vedere al dimensiunii și vitezei), putere a unui singur nucleu, cât și scalabilitatea către sisteme multi-nucleu.

Stațiile de lucru multi-nucleu moderne sunt în mod fundamental dezvoltate din chip-uri mono-nucleu, grupate după criterii geometrice. Aceste soluții oferă performanțe mono-nucleu foarte bune, dar nu scalează bine în zona miilor de nuclee. Acestea consumă de obicei multă putere, și progrese recente pentru a combate aceasta problemă sunt în direcția de a combina nuclee orientate spre performanță cu cele orientate spre economie energetică. O direcție recentă pentru aceste mașini este adăugarea extensiilor de calcul matriceal, pe lângă extensiile pentru calcul vectorial.

Cele mai uzuale soluții pentru accelerarea calculului sunt plăcile video, folosite în contextul calculului de uz general. Fiind un circuit orientat către grafică, acesta nu se pretează în mod natural calculului de uz general. Pe de altă parte, datorită accesului facil la dispozitive și efortului depus de Nvidia și AMD, s-au făcut îmbunătățiri mari privind folosirea acestei tehnologii pentru computație de multiple feluri. Datorită omniprezenței sistemelor de învățare automată din peisajul computației moderne, plăcile video din generația curentă au adăugat structuri fizice specializate pentru cele mai folosite funcții (nuclee de calcul tensorial). Nucleele tensoriale au devenit o componentă standard al plăcilor grafice moderne, Nvidia oferind tutoriale și sfaturi privind utilizarea lor, [1], fiind evaluate din multiple perspective, cum ar fi performanța [2] [3] și comportamentul numeric [2] [4]. Folosind nuclee tensoriale, se îmbunătățește eficiența energetică pentru aplicații de IA (pentru care au fost și proiectate), dar și potențial în alte cazuri (de exemplu [5]). Tehnologia AMD se numește "Matrix core" iar cea de la Nvidia "Tensor core".

Fiind soluția cea mai uzuala pentru accelerație, plăcile video au avut impact asupra tuturor domeniilor de calcul. Comunitatea științifică a tratat acest subiect din multiple puncte de vedere: calcul propriu-zis, gestionarea memoriei și a transferurilor, reprezentarea datelor și multe altele. Tehnicile de calcul eterogen pot fi observate în [6] și tehnici de optimizare pentru placi grafice în [7].

Când și mai multă performanța este dorită se pot folosii structuri fizice personalizate implementate in FPGA pentru calculul intens. Următorul pas constă în accelerarea cu structuri fizice implementate în siliciu, un bun exemplu al acestei abordări fiind, TPU ("Tensor Processing Unit").

1.2 Scopul tezei

Această teză se concentrează pe îmbunătățirea acceleratorului "many-core" de uz general "Connex" și pe dezvoltarea unui nou sistem de calcul paralel având ca țintă implementarea pe FPGA. Acceleratorul este proiectat să se lege cât mai strâns la un calculator gazdă pentru a putea prelua procesele de calcul intense ale acestuia. Pentru a valida îmbunătățirile propuse, algoritmi care le folosesc au fost scriși si testați.

Îmbunătățirile arhitecturale vor include înlocuirea rețelei de reducere cu o rețea de tip Prefixe-Permutare-Grupare, rescrierea acceleratorului pentru a-l face complet parametrizabil și reconfigurabil, generalizarea interfeței de IO, adăugarea de suport pentru numere în virgulă mobilă și multe altele.

Algoritmi principali dezvoltați pentru aceasta mașină vor fi: AES, transpusa pentru matrice pătrată și transformata Fourier rapidă, (FFT) în diferite configurații.

1.3 Conținutul tezei

Această teză detaliază munca depusă pentru îmbunătățirea arhitecturii acceleratorului paralel many-core de tip Map-Reduce. Versiunea precedenta a acceleratorului este detaliată in Capitolul 2.

Capitolul 3 explorează implementarea rețelei de Prefixe-Permutare-Grupare, adăugată pentru a include rețeaua de Reducere, dar și a oferi noi funcționalități.

Capitolul 4 explorează implementarea parametrizării și reconfigurabilității, așa cum au fost folosite în această arhitectură.

Capitolul 5 are în vedere reconfigrabilitatea pentru îmbunătățirea IO.

Capitolul 6 și Capitolul 7 prezintă îmbunătățirea puterii de calcul din celule prin adăugarea unui mecanism de stivă și a suportului pentru calcul în virgulă mobilă.

Diverse îmbunătățiri hardware mai mici sunt prezentate în Capitolul 8 și modificări ale setului de instrucțiuni și ale structurii acestuia apar în Capitolul 9.

Rezultatele sintezei pentru FPGA sunt prezentate în Capitolul 10.

Capitolul 11, Capitolul 12, Capitolul 13 prezintă îmbunătățiri la nivelul de aplicație, privind AES, Transpunerea și FFT.

Un al doilea rând de îmbunătățiri minore sunt prezentate în Capitolul 14, de data asta privind testarea, sistemul și dezvoltarea de software.

Capitolul 15 prezintă concluziile și prezintă posibile dezvoltări viitoare.

Capitolul 2
Forma anterioară a acceleratorului de tip Map-Reduce

2.1 Considerente teoretice

Spre deosebire de mașinile seriale, hardware-ul pentru calcul paralel a avut un complet alt drum de dezvoltare. Mașinile seriale au avut un curs de dezvoltare normal, format din următoarele etape: (1) model computațional - mașina Turing [8], (2) model abstract - modelul Harvard/Von Neumann, (3) dezvoltarea comercială și manufactură la scară largă, (4) dezvoltări arhitecturale x86, ARM, etc.

Pe de alta parte, mașinile paralele au urmat un cu totul alt drum, începându-și viața ca o grupare ad-hoc de mașini seriale. Modele abstracte au fost ulterior dezvoltate, dar modelul computațional nu a primit atenția cuvenită. Există totuși un model computațional care se potrivește cu abordarea paralelă, modelul Kleene al funcțiilor parțial recursive [9]. Bazându-se pe acest model, nucleul arhitecturii acceleratorului Connex a fost definit și implementat [10], [11], [12].

Structura fizică computaționala este definită de regula de compoziție a modelului matematic, așa cum se poate observa din lucrările de mai sus, astfel apare modelul abstract Map-Reduce.

2.2 Nucleul acceleratorului

Pornind de la modelul abstract, arhitectura acceleratorului se bazează pe două categorii de operații:

- Operații Map care primesc un vector de intrare și întorc un vector de ieșire, aplicând aceeași operație asupra tuturor datelor;
- Operații de Reducere care primesc un vector de intrare și au ca rezultat un scalar.

Operațiile de tip Map sunt executate într-un vector de mașini de calcul simple, numite colectiv "Array". Operațiile de Reducere sunt executate printr-o rețea logaritmică de celule foarte simple, având o structură arborescentă. Rețeaua de reducere primește date și comenzi (funcția pe care să o execute) de la Array. Pentru a controla operațiile din Array, o a treia structură este necesară, "Controller"-ul. Structura fizică ce rezultă este prezentată în Figura 2.1.

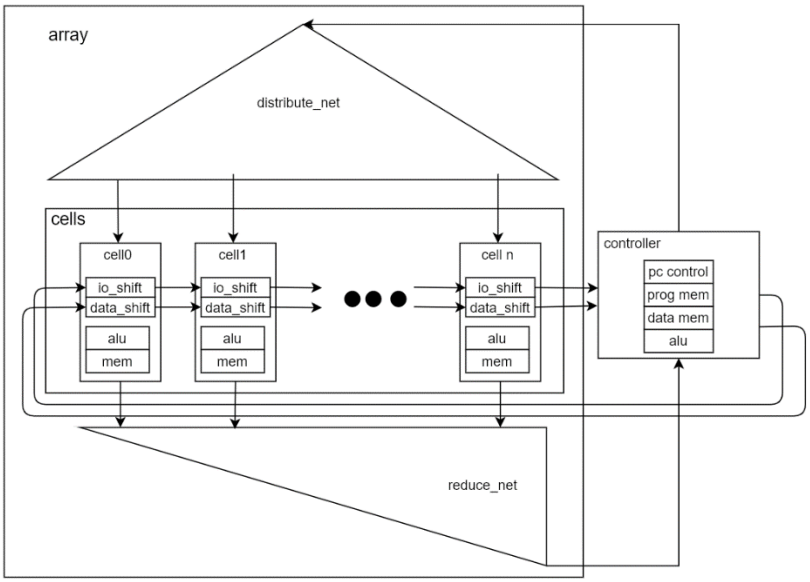

Figura 2.1 Structura nucleului original al acceleratorului Connex (adaptat din [12])

2.3 Arhitectura setului de instrucțiuni (ISA)

O instrucțiune completă (32b) este alcătuită din două instrucțiuni normale în limbaj de asamblare (ASM), fiecare având 16b: una este executată în Controller și cealaltă în fiecare celulă activă din Array.

Fiecare instrucțiune este alcătuită din trei componente:

- Codul operației (5b) -folosit în selecția operației dorite;
- Operand (3b) - folosit în selecția operandului care va fi folosit împreună cu acumulatorul;
- Valoare (8b) - valoare scalară care poate fi interpretată ca operand.

O descriere detaliată a tuturor operațiilor poate fi găsită în [13], capitolul 2.1.3.

2.4 Sistemul cu accelerator - hardware

Sistemul cu accelerator a fost dezvoltat în cadrul [13]. Este alcătuit din următoarele structuri principale: Accelerator, ARM Cortex-A9 (Sistem de procesare), folosit ca gazdă și un modul de DMA.

2.5 Sistemul cu accelerator - software

Pentru ca sistemul hardware să poată fi folosit, o stivă software corespunzătoare este necesară. Colegii mei au dezvoltat o astfel de stivă, bazată pe kernel-ul Linux PYNQ ce se găsește la [14]. Stiva software este alcătuită din: interfață hardware, kernel Linux, sistem PYNQ, Jupyter Notebook și mediu de dezvoltare Python.

Folosind acest sistem, operații de algebra lineara au fost rulate pe accelerator. Datele au fost generate din mediul Python, trimise către accelerator (împreună cu comenzile, procesate în accelerator și apoi extrase și vizualizate.

Capitolul 3
Îmbunătățire arhitecturală: Rețeaua de Prefixe-Permutare-Grupare-Reducere

Acest capitol descrie proiectarea, implementarea, testarea și integrarea unei rețele multifuncționale de calcul, descrisă în [16] și în [17].

3.1 Prefixe și Reducere

Operațiile de tip Reducere primesc un vector de intrare și întorc un scalar la ieșire. Pe de alta parte, operațiile de tip Prefix, primesc un vector și întorc tot un vector.

Ieșirea finala a rețelei de prefixe este chiar rezultatul operației de Reducere asupra datelor de la intrare.

3.2 Rețeaua de permutare Beneš-Wacksman

Structura de circuit considerată inițial a fost cea a rețelei de permutare proiectată de Vaclav E. Benes [18] și optimizată de Abraham Waksman [19]. Acest circuit permite toate permutările datelor de la intrare să fie efectuate, daca există un set de biți de control corespunzători. Un exemplu de astfel de rețea se poate vedea în Figura 3.1. Colorate cu roșu sunt celulele nenecesare (așa cum a fost demonstrat de Abraham Waksman în [19]) care vor fi regiștri pipeline fără multiplexoarele ce permit permutarea. Colorat în verde, sunt straturile de intrare și de ieșire iar în albastru și galben, cele două subrețele, conform cu definiția recursivă. Fiecare celulă este alcătuită din două multiplexoare, primind câte doi operanzi la intrare și având două rezultate la ieșire.

3.3 Funcții implementate

Următoarele funcții au fost implementate în rețeaua finală: reducere-adunare, reducere-minim, reducere-maxim, reducere-și, reducere-sau, reducere-xor, prefixe-adunare, prefixe-xor, permutare, grupare.

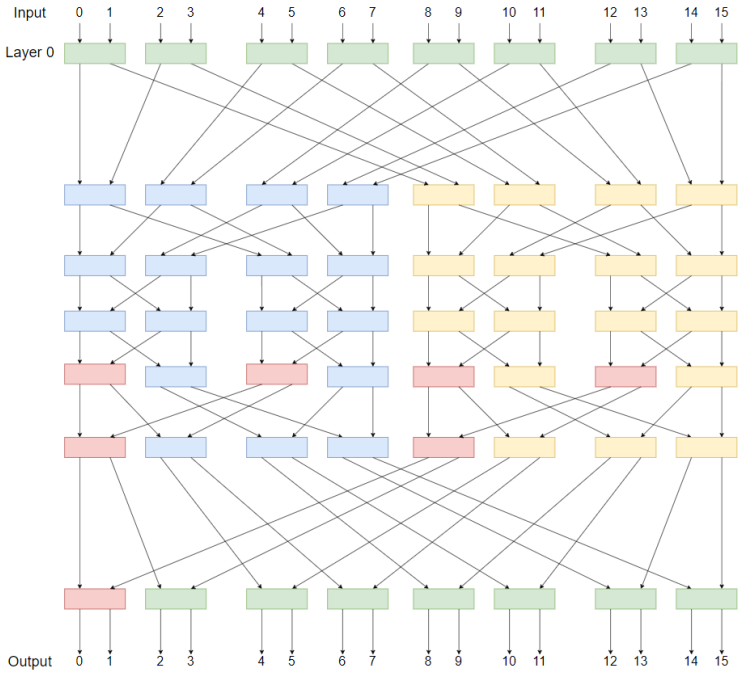

Figura 3.1 Exemplu de circuit de permutare Beneš-Wacksman cu 16 intrări

3.4 Circuitul multifuncțional Prefixe-Permutare-Grupare-Reducere: descriere hardware

Pornind de la forma și structura circuitului de permutare Beneš-Wacksman, am dezvoltat un nou circuit ce permite multiple alte operații. Acest circuit a fost publicat în [16] și suplimentar în [17]. Deși forma rețelei a rămas nemodificată, structura internă a celulei a fost semnificativ modificată ca să permită noile funcționalități. Structura internă a celulei multifuncționale este prezentată în Figura 3.2.

În funcție de poziția celulei și de funcțiile pe care rețeaua trebuie să le poată îndeplini, structura internă a celulelor poate suferi modificări semnificative. Aceste variații se manifestă prin complexitatea blocului de control și prezența sau absența unor blocuri hardware.

Cele două operații de rearanjare a datelor (permutarea și gruparea), necesită un bit de control care va funcționa ca selecție pentru cele două multiplexoare interne din fiecare celulă. Fiecare celulă necesită un astfel de bit de control pentru a funcționa corect. Cum rețeaua e pipeline și pentru a reduce complexitatea fizică a firelor de legătură, biții de control se propagă împreună cu datele pe care le ghidează, fiecare celulă va consuma un bit de control.

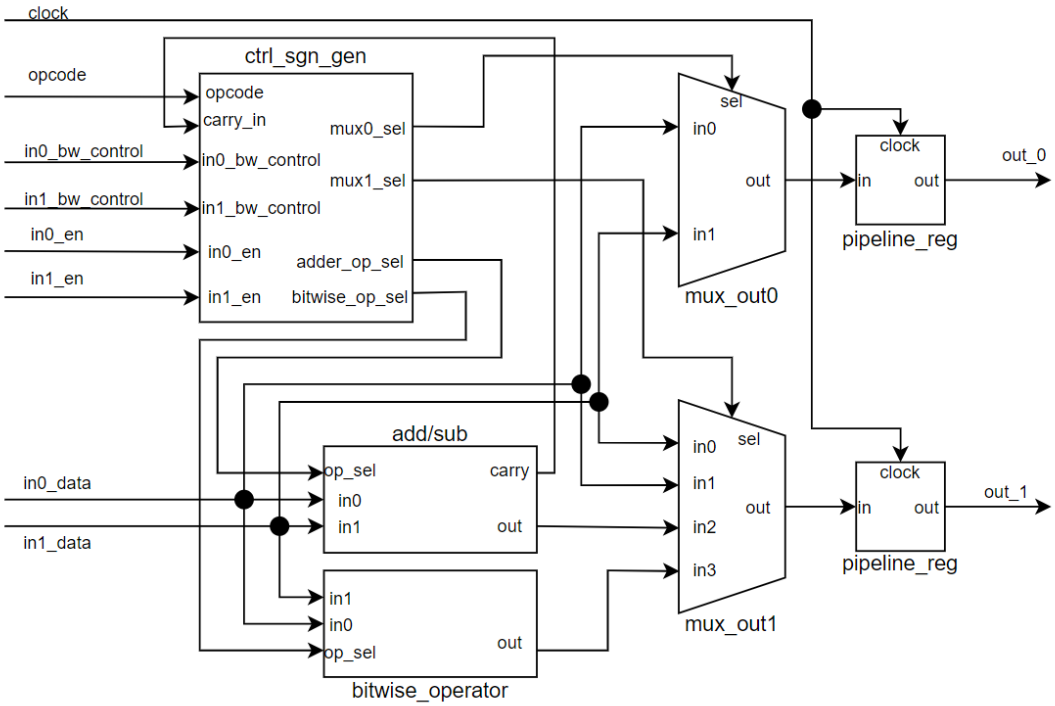

Figura 3.2 Exemplu de structură internă a celulei multifuncționale

3.5 Versiunea secvențială

O versiune secvențială a acestei rețele a fost proiectată si testată, folosind un singur strat de celule de permutare/calcul și câte doua multiplexoare cu multe intrări pentru fiecare celulă. Această versiune se poate folosii atunci când spațiul este mai important decât performanța. În accelerator, această versiune a fost abandonată.

3.6 Generarea biților de control

Generarea biților de control este o operație complexă. Ea este împărțită în două etape: generarea biților de control pentru fiecare celulă și asamblarea biților de control (pornind de la sfârșit spre început, în funcție de drumul datelor prin rețea) pentru a obține cuvinte de control. Cod în C++ a fost scris pentru a efectua aceste operații.

3.7 Testare

Pentru a valida corecta funcționare a circuitului, un modul de test a fost scris în SystemVerilog. Modulul de test generează datele de intrare, trimite datele atât către rețea cât și către un model ideal nesintetizabil al acesteia și la final compară ieșirile celor doua. Toate funcțiile au fost testate, inclusiv tranzițiile dintre acestea. Circuitul a trecut toate testele propuse.

3.8 Rezultatele sintezei

Rezultatele de sinteză pentru rețeaua multifuncțională sunt prezentate în tabelul 3.1. Sinteza a fost făcută pentru o placă PYNQ-Z2 având un ZYNQ7020 SoC.

Tabelul 3.1 Rezultatele sintezei rețelei multifuncționale

Nr intrări	Nr celule	LUT-uri	Registre	Medie LUT-uri/celulă	Medie regiștri/celulă
8	20	1240	1459	62	73
16	56	3813	3921	68	70
32	144	10208	10368	71	72
64	352	25894	26252	74	75
128	832	63389	64146	76	77
256	1920	151180	152627	79	79

3.9 Integrarea în acceleratorul Map-Reduce. Arhitectura Map-Prefixe-Permutare-Grupare-Reducere

Această rețea a fost proiectată având în minte acceleratorul descris în Capitolul 2. Aceasta va veni ca o îmbunătățire pentru rețeaua de Reducere deja prezentă. Astfel rețeaua de Reducere se conectează direct în Controller și toate ieșirile rețelei de permutare/prefixe se conectează înapoi la celulele de calcul.

3.10 Concluzii

În acest capitol proiectarea rețelei multifuncționale (Prefixe, Permutare, Grupare, Reducere) a fost prezentată. Acest proces a implicat proiectarea (pornind de la rețeaua de permutare Beneš-Waksman), deciderea și implementarea a multiple funcționalități, testarea tuturor funcțiilor dezvoltate și integrarea în acceleratorul final.

Rețeaua poate acum face multiple operații (așa cum sunt descrise în Capitolul 3.3), inclusiv toate operațiile pe care rețeaua anterioară de reducere le putea efectua.

Folosirea macrou-urilor și a parametrizări ajută acest circuit să aibă un grad ridicat de flexibilitate, care la rândul ei a ajutat integrarea în accelerator.

Pentru operația de permutare există un dezavantaj, anume faptul ca biții de control trebuie calculați în avans și operația aceasta este una dificilă.

Un al doilea dezavantaj, deși unul mult mai mic, este faptul ca operația de grupare necesita o sumă de prefixe peste biții ce specifică dacă celulele sunt active, înainte de a putea fi folosită.

În mod similar, alte funcții asociative, cu două variabile ce suportă inverse, pot fi implementate folosind această formă de rețea, deși în unele cazuri, circuitele poate nu sunt cea mai eficientă implementare a oricărei funcții.

Capitolul 4
Îmbunătățire arhitecturală: Parametrizare și reconfigurabilitate

4.1 Rescriere și parametrizare

Felul în care era scris inițial codul acceleratorului, deși funcțional, nu se conforma cu bunele practici moderne privind dezvoltarea software/hardware. Din acest motiv, tot codul a trebuit sa fie rescris într-un format mai ușor de urmărit și utilizat.

Cu această ocazie a apărut și o oportunitate pentru parametrizare și reconfigurabilitate. Prin parametrizare mă refer la macro-uri ce modifică structura fizică a mașinii. Parametrii ce se referă la numărări (de exemplu cele din ISA ce ajută la numerotarea instrucțiunilor) și macro-uri ce pot fi generate din alte macro-uri nu se consideră parametrii.

Cei mai importanți parametrii sunt cei ce se referă la numărul de celule, la cantitatea de memorie din fiecare celulă și la prezența sau absența unor blocuri hardware.

4.2 Reconfigurabilitate hardware și sinteză condițională

Multe blocuri hardware au fost integrate în accelerator într-un fel care să permită sinteza condițională. În funcție de o multitudine de factori, codul este scris în așa fel încât să poată oferi flexibilitate privind capabilități hardware avansate și dimensiunea și complexitatea circuitului rezultat.

Considerăm reconfigurabilitatea ca un concept deosebit de important adus de creșterea FPGA-urilor ca dimensiuni și folosire. Pe arhitectura noastră, acest lucru este foarte important deoarece permite potrivirea arhitecturii cu aplicații mai specifice, păstrând în același timp numele de accelerator de uz general. În contextul unei aplicații, acest accelerator poate să obțină rezultate bune nefiind complet reconfigurabil (înlocuirea completă în FPGA a acceleratorului în timpul rulării ar fi prea lentă) și folosind o cantitate de resurse hardware mai mică decât a avea multiple acceleratoare dedicate fiecărei subfuncții.

Capitolul 5
Îmbunătățire arhitecturală: Generalizarea IO

5.1 Starea anterioară a interfeței de IO

Atât interfața de intrare cât și interfața de ieșire sunt pe 64 de biți, având semnale suplimentare pentru control și pentru mecanismul de propagare. Cum fiecare celula primește 32 de biți de date, două celule pot fi servite în fiecare moment de timp, astfel apare "celula duala". O îmbunătățire suplimentara (prezentată în [13]), creează quad-celula, pentru a putea avea un debit total de 64 de biți per ciclu. Figura 5.1 exemplifică acest concept

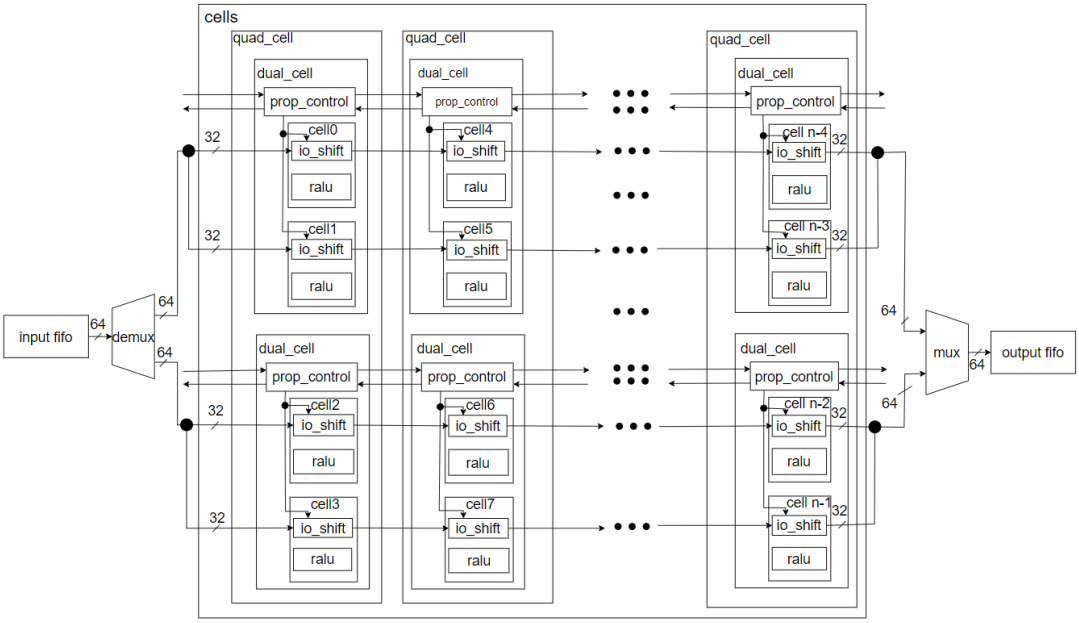

Figura 5.1 IO: organizarea quad-celula

5.2 Îmbunătățiri

Deși forma propusa in Figura 5.1 este atât funcțională cât și eficientă, pentru a păstra conceptele de parametrizare și reconfigurabilitate prezentate în Capitolul 4, a trebuit să

generalizez acest concept. Astfel a apărut structura de multicelulă și multicelulă parțială (a se vedea Figura 5.2).

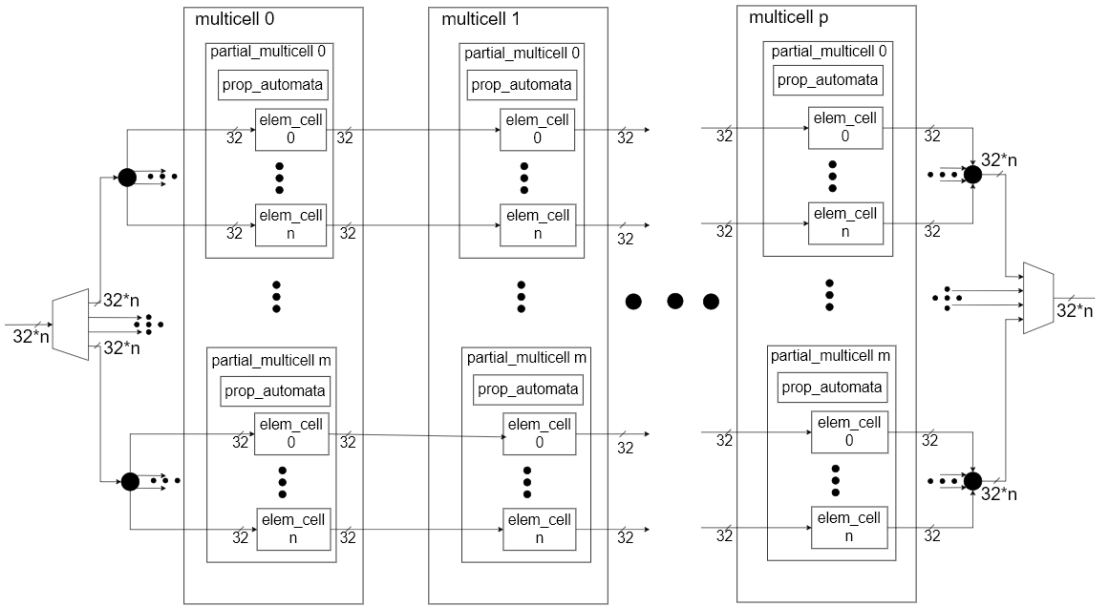

Figura 5.2 IO: organizarea multicelulă

Celulele elementare sunt grupate în multicelule parțiale. Toate celulele din cadrul unei multicelule parțiale sunt încărcate în paralel. Multiple Parțiale formează o multicelulă. În cadrul unei multicelule, toate Parțialele ce o alcătuiesc sunt încărcate sau descărcate secvențial, una față de alta. Astfel, multicelulele parțiale devin o generalizare a celulelor duale iar multicelulele o generalizare a quad-celulelor. Aceasta generalizare va ajuta cu interfețe late de date la nivelul sistemului.

5.3 Concluzii

Deși pentru sistemul actual această generalizare nu este absolut necesară (interfața de tip AXI Stream are maxim 64 biți), având în vedere perspectiva altor interfețe cu un număr posibil mai mare de biți, consider că aceasta îmbunătățire este importantă.

Secundar, această flexibilitate poate fi folosită în cazul FPGA pentru a ajuta algoritmii de plasare și rutare. Deși netestat, teoretizez că pentru același număr de celule, unele aranjamente pot fi mai benefice din punctul de vedere al vitezei ceasului.

Suplimentar, această organizare a permis o parte din opțiunile privind poziționarea decodorului pentru instrucțiunile celulelor (așa cum este explicat în Capitolul 8.3).

Capitolul 6
Îmbunătățire arhitecturală: Procesor cu acumulator și stivă

6.1 Introducere

Procesoarele aparțin de trei categorii majore:

- Cu regiștri - operațiile pot avea loc între oricare doi regiștri;
- Cu acumulator - operațiile au loc între acumulator și un al doilea operand;
- Cu stivă - operațiile au loc între vârful stivei și un al doilea operand de pe stivă, în mod uzual datele ce sunt imediat sub acesta.

Pe mașina noastră, varianta cu regiștri a fost abandonată deoarece formatul ISA era inițial prea mic pentru a conține codul operației împreună cu adrese pentru un număr considerabil de regiștri de uz general adresabili. Din acest motiv cât și din cauză că performanțele nu au fost afectate negativ în mod semnificativ, s-a ales varianta cu acumulator si un registru foarte mare folosit ca al doilea operand.

Procesoarele cu acumulator eliberează spațiu în formatul instrucțiunilor, dar din păcate necesită un număr mare de operații de încărcare/descărcare. Pe de altă parte, un procesor doar cu stivă este foarte restrictiv privind al doilea operand (ce trebuie să se afle tot pe stivă), dar este și mai economic din punct de vedere al formatului instrucțiunii. Operațiile sunt efectuate între vârful stivei și sub-vârful stivei. O operație obișnuită consuma ambii operanzi și plasează rezultatul în vârful stivei. Se poate vedea ca două operații de tip "pop" urmate de o operați de tip "push" cu rezultatul.

6.2 Implementare

Un procesor cu stivă nu este prin definiție mai bun sau mai rău decât un procesor cu acumulator și pentru a combina avantajele celor doua abordări, acestea au fost combinate într-o mașina hibridă eficientă.

Abordarea pe bază de acumulator s-a dovedit foarte eficientă pentru acest accelerator pe multiple clase de probleme și nu aveam motive să renunță mal ea. Mecanismul cu stivă permite păstrarea datelor pentru uz ulterior, într-o manieră independentă de adrese. Structura hardware rezultată este prezentată în Figura 6.1. Operațiile cu stiva implementate sunt prezentate în Figura 6.2.

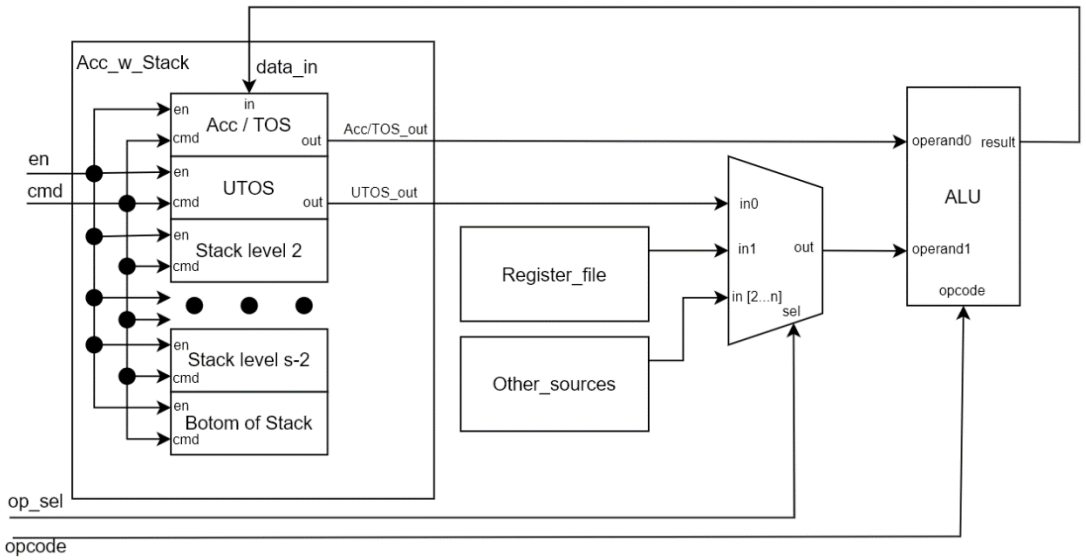

Figura 6.1 Design hardware al procesorului cu stivă

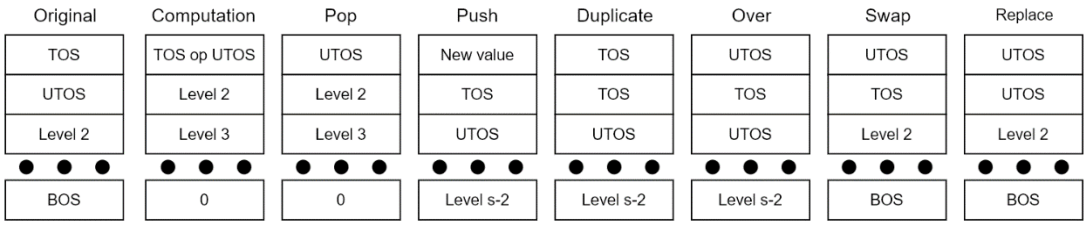

Figura 6.2 Operații cu stiva

Această îmbunătățire poate fi văzută și ca adăugarea unui nou mod de adresare pentru selecția operandului.

Testarea arhitecturii s-a dovedit de succes, găsind multiple cazuri în care lucrul cu stiva a ajutat timpul de execuție. În cazurile găsite până acum, s-a dovedit potrivită pentru optimizări mici, eliminând câte una sau doua instrucțiuni fața de varianta doar cu acumulator. Pe parcursul unor programe mai mari sau în cazul buclelor imbricate, astfel de optimizări mici pot sa devină semnificative.

Testare suplimentară este necesară pentru a măsura și judeca impactul acestei îmbunătățiri asupra arhitecturii.

Capitolul 7
Îmbunătățire arhitecturală: Suport pentru calcul în virgulă mobilă

7.1 Descrierea operațiilor în virgulă mobilă

Suport pentru operațiile în virgulă mobilă a fost adăugat la arhitectura acceleratorului.

Toate operațiile în virgulă mobilă sunt gândite ca microprograme. Toate microprogramele necesită un anumit număr de cicli de ceas (a se vedea Tabelul 7.1) pentru a fi efectuate. Cum unitatea de calcul în virgulă mobilă nu este un coprocesor separat, alte calcule nu se pot face cât timp aceste operații se fac. Operații cu numere denormalizate, 0, infinit, NaN sunt de asemenea suportate.

Tabelul 7.1 Operații în virgulă mobilă - cicli de ceas per operație

adunare	7
scădere	7
înmulțire	7
împărțire	57

7.2 Hardware suplimentar pentru calcule în virgulă mobilă

Atât în "Array" cât și în "Controller", hardware-ul adăugat este identic.

Pentru a respecta conceptul de reconfigurabilitate și sinteza condiționată, suportul hardware pentru calcul în virgulă mobilă este controlat prin doi parametrii (unul pentru "Array" și unul pentru "Controller"). Acești parametrii controlează existența operațiilor în virgulă mobilă și prezența sau absența circuitelor suplimentare.

Din punct de vedere hardware, aceste operații necesită un modul de rotire/shift-are și un bloc dedicat pentru calcul în virgulă mobilă. Acesta conține relativ multă logică suplimentară, fiind aproximativ la fel de mare ca restul celulei (excluzând primitiva DSP folosită ca ALU). Dimensiunea ridicată va trebui sa fie optimizată pe viitor.

7.3 Concluzii

Operații în virgulă mobilă au fost implementate și testate atât pentru "Array" cât și pentru "Controller".

Aceste operații au fost implementate ca microprograme, astfel încât celulele sa nu necesite un coprocesor pentru virgulă mobilă. Astfel, resurse hardware deja prezente au putut fi refolosite.

Hardware suplimentar a trebuit sa fie proiectat și integrat în accelerator astfel încât operațiile să funcționeze. Unitate aritmetică și logică și registrul local de shift-are au fost folosite în cadrul microprogramelor pentru a reduce hardware-ul suplimentar folosit.

Operații cu numere denormalizate, 0, infinit, NaN sunt de asemenea suportate.

Marele dezavantaj al implementării curente este numărul mare de LUT-uri ocupate.

Capitolul 8
Îmbunătățiri arhitecturale: Îmbunătățiri diverse

8.1 Rotiri globale prin registrul de shift-are global

Registrul de shift-are global (RSG) este folosit pentru a muta date între celule învecinate, inițial doar către dreapta. RSG-ul este distribuit de-a lungul acceleratorului, o celulă de shift-are per celulă principală de calcul. Versiunea curentă de RSG permite comunicarea cu vecinul din dreapta, din stânga și cu acumulatorul celulei de calcul de care aparține. Conexiunea cu acumulatorul este bidirecțională.

Pentru a folosii mai bine această resursă, multiple îmbunătățiri au fost făcute. În primul rând shift-area bidirecțională a fost implementată. Elementele de memorare au rămas aceleași, dar un multiplexor suplimentar a fost adăugat. Cu această structură hardware, operații de rotire au fost de asemenea implementate. Structura rezultată este descrisă în figura 8.1.

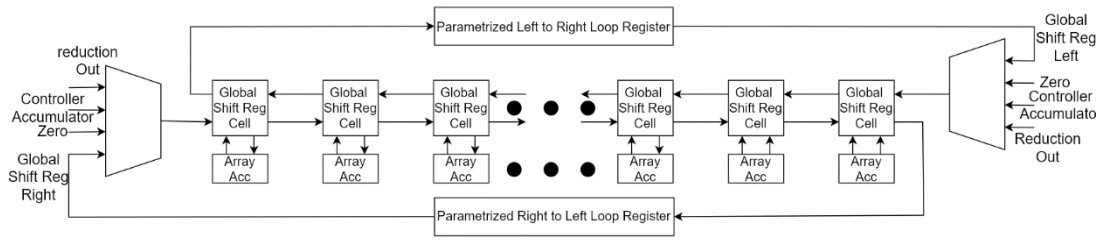

Figura 8.1 Registrul global de shift-are - varianta îmbunătățită

Instrucțiuni suplimentare au fost de asemenea adăugate.

8.2 "Barrel shifter" pentru shift/rotire locală

Operațiile de Shift/Rotire pot să aibă loc și la nivelul celulelor, folosind data ce se găsește în acumulator. Acest fel de shift-are este foarte util pentru multipli algoritmi și a fost implementat atât în celulele de calcul cât și în "Controller". Aceste operații au necesitat adăugarea unui "Barrel shifter" (pentru rotiri) și a unui generator de bară urmat de porți "și" (pentru operații de shift-are).

8.3 Locația decodorului pentru celule

Inițial decodarea instrucțiunilor pentru celule se întâmpla la nivelul acestora. Acum aceasta se poate întâmpla în una din patru locații: (1) "Controller", un decodor pentru toate celulele; (2) nivelul de multicelulă; (3) nivelul de multicelulă parțială; (4) nivelul celulei elementare, un decodor per celulă. Compromisul prezent aici este între consumul de arie (multiple decodoare rezultă într-o arie mai mare) și o congestie mai mare prin rețeaua de distribuție (daca decodarea are loc doar într-un loc și instrucțiunile sunt expandate în comenzi ce necesită fire de legătura). Studii suplimentare sunt necesare pentru a vedea care abordare este optimă și cum se definește optimalitatea în această privință.

8.4 DMA pentru memoria din "controller"

Deși nu la fel de des folosit ca mecanismul DMA pentru memoria din celule, colegii mei și cu mine am decis să implementăm un DMA și pentru memoria din "Controller". Acest mecanism este util când cantități relativ mari de date sunt trimise către "Controller" și folosirea căii de program pentru parametrii de funcție ar fi prea lentă. Astfel de cazuri includ filtrele folosite la convoluții sau chei calculate în avans pentru algoritmi criptografici. Acest mecanism funcționează foarte similar cu cel din celulele de calcul, descris în [13] și [15].

8.5 Adăugarea de regiștri de adresare

Atât "Controller"-ul cât și "Array"-ul suporta adresare indirectă bazată pe registrul AddrReg. Acesta ține adresa de bază la care se adaugă "offset"-ul pentru a obține adresa fizică finală. Acest mecanism era inițial identic în toate structurile.

Modurile de adresare indirecte sunt valoroase din doua motive. În primul rând, unele programe (majoritatea calculului intens) folosește tipare de adresare relativ clare, ce sunt accelerate de acest mecanism, în special de adresarea indirecta cu AddrReg și incrementare. În al doilea rând, adresa de bază pentru datele salvate de fiecare program ce rulează, permite transferului să aibă loc în timp ce cealaltă jumătate este folosită pentru calcul. Folosind această abordare, orice adresare trebuie să fie făcută în mode relativ în cadrul tuturor programelor scrise.

O problemă apare când calculele au multiple tipare de acces, fiecare cu o regulă proprie. O a doua problemă apare când multiple funcții primitive vor să proceseze date, dar prima modifică adresa de bază. În această situație, următoarele apeluri vor folosii o adresă de bază incorectă. În "Controller", am implementat cea mai flexibilă (și consumatoare de spațiu) soluție. Aceasta constă în a avea multipli regiștri pentru adresa de bază și adăugarea unei instrucțiuni pentru schimbare între aceștia.

În "Array", cum fiecare celulă are un AddrReg, a avea un bloc de regiștri și logica necesară controlului acestora ar ocupa prea mult spațiu. O structură simplă cu o stivă cu două nivele a fost folosită.

Capitolul 9
Îmbunătățiri arhitectural: ISA

9.1 Noul format al instrucțiunilor

Prima și cea mai notabilă îmbunătățire a fost creșterea dimensiunii unei instrucțiuni de la 16 la 32 de biți. Dimensiunea codului de operație a rămas la fel, dar fiind mai mulți biți pentru "valoare", ajută cu creșterea dimensiunilor tuturor datelor.

Acest format de instrucțiune a fost scris în mod parametrizat, astfel încât creșteri ulterioare să poată fi făcute cu ușurință. Macro-uri au fost folosite oriunde a fost nevoie de informație privind formatul și dimensiunea instrucțiunii.

9.2 Înțelesul și organizarea codurilor de operație

Toate codurile de operație sunt generate folosind macro-uri astfel încât numerotarea și asignarea valorilor să aibă loc automat în etapa de elaborare.

Pentru fiecare familie de coduri de operație (control sau operații cu date), locația unde instrucțiunile sunt executate afectează de asemenea numerotarea. Același cod de operație poate fi folosit în scopuri diferite in "Array" sau în "Controller".

Un cod de operație poate avea până la patru înțelesuri diferite, în funcție de tipul operandului și de locația unde acesta va fi executat. Atât asamblorul cât și programul de sinteză hardware folosesc aceleași fișiere cu macro-uri pentru a se asigura interoperabilitatea dintre programe și unelte.

9.3 Compresia codului de instrucțiune

În procesul de îmbunătățire al arhitecturii, instrucțiuni noi au trebuit să fie adăugate, iar la un moment dat, nu au mai rămas coduri de instrucțiune libere pentru acestea. Soluția adoptată a fost să grupez codurile de instrucțiune al instrucțiunilor cu rol similare.

Original, fiecare instrucțiune ASM avea propriul cod, lucru care s-a dovedit problematic mai ales pentru categoria instrucțiunilor de control. Pentru a rezolva această problemă, instrucțiunile similare au fost grupate sub același cod de instrucțiune și o parte din câmpul pentru "valoare" a fost folosit ca o continuare a codului de instrucțiune. Această metodă a fost folosită pentru instrucțiuni de salt, de control a stivei, de selecție spațială, de shift-are cu valoare fixă, de shift-are prin registrul global de shift, de încărcare/descărcare și altele

9.4 Salturi

Două îmbunătățiri principale au fost făcute pentru instrucțiunile de salt: adăugarea unui registru pentru decrementare și a unui registru pentru comparație cu valoare. Inițial, toate salturile putea face comparație doar cu acumulatorul.

Registrul de decrementare este folosit în cazul instrucțiunilor de salt cu decrementare și permite atât decrementarea cât și incrementarea cu orice valoare presetabilă. Trebuie menționat ca valoarea acestui registru este mereu scăzută, dar scăderea poate fi făcută și cu un număr negativ, astfel obținându-se incrementarea. Acest registru este extrem de valoros deoarece permite scrierea unui cod ASM mult mai curat și mai ușor de înțeles. În unele cazuri, contorul de buclă "i" este folosit și în calculul de adrese sau pentru alte informații utile pentru "Array" și a putea număra în ambele direcții și la nevoie cu salt devine foarte util. Acesta permite acces la memorie cu pas diferit de 1, fără a necesita pași suplimentari în "Controller".

Toate tipurile anterioare de salturi foloseau acumulatorul și fanioanele acestuia: zero, negativ, depășire. Pentru flexibilitate maximă privind modul de scriere al buclelor, am dorit să iau în considerare și cazul "daca(acumulator == valoare). Astfel, "registrul de valoare" a apărut. Atât cazul de egalitate cât și de inegalitate sunt luate în calcul. fiecare având instrucțiunea ei de ASM. O îmbunătățire secundară a fost făcută care să permită acestui registru să fie incrementat sau decrementat de către registrul de decrementare prezentat anterior și adăugarea de fanioane și pentru acesta. Astfel se obține un nivel suplimentar de paralelism in "Controller", prin decuplarea acumulatorului de logica necesară pentru controlul buclelor. Un beneficiu suplimentar care a fost adăugat a fost punerea unui număr parametrizabil de astfel de regiștri de control, care permite un control foarte fin și eficient mai ales în cazul buclelor imbricate.

9.5 Schimb între acumulator și o locație de memorie

O nouă instrucțiune a fost adăugată în ISA care să permită interschimbarea dintre acumulator și o locație de memorie dorită. Aceasta poate fi văzută ca o combinație între o încărcare și o descărcare, ambele având loc în același ciclu de ceas.

Adăugarea acestei instrucțiunii a necesitat schimbări foarte mici în hardware. Decodorul de instrucțiune a trebuit un pic modificat, căile de date fiind deja implementate. Această instrucțiune este utilă ca o optimizare pentru viteză în unele situații.

9.6 Shift/Rotire cu dimensiune fixă

În arhitectura originală, doar operații de shift la dreapta cu o unitate se puteau efectua. Shift la stânga se putea face prin înmulțire cu puteri ale lui doi. Pentru unele aplicații, acest lucru s-a dovedit un factor limitant.

Pentru multe aplicații, shift-are necesară este cu o cantitate fixă, de obicei cu 8 biți. Deoarece această operație este foarte comună și modificările hardware necesare au fost minore (pe lângă accelerația pe care o generează), o instrucțiune noua de shift cu

dimensiune fixă a fost adăugată. Totul legat de acest mecanism este parametrizabil la sinteză, atât prezența sau absența sa cât și valoarea fixă folosită pentru shift-are..

9.7 Shift/Rotire globala cu sau fără registrul de activare

Registrul de shift-are global opera independent de starea celulelor, mai specific, nu lua în calcul dacă celule erau active sau nu. Am proiectat și implementat trei variante de shift-are pentru registrul global, toate în ambele direcții.

- **Shift fără să țină cont de stare**
  Starea celulei nu este luată în calcul, toate celulele efectuând operația.
- **Shift care ține cont de stare și adaugă "0"**
  Dacă celula curentă este inactivă, valoarea ei rămâne neschimbată. Dacă este activă și celula precedentă este inactivă, valoare din celula anterioară este ignorată și un "0" este inserat.
- **Shift care ține cont de stare și păstrează valoarea.**
  Dacă celula curentă este inactivă, valoarea ei rămâne neschimbată. Dacă este activă și celula precedentă este inactivă, valoarea celulei precedente este ignorată și valoarea celulei curente este menținută. Practic, se vor duplica valorile celulelor active la granița cu celule inactive.

Din punct de vedere hardware, multiplexor intern al fiecărei celule din registrul de shift-are globală a crescut, acesta fiind singurul dezavantaj al adăugării acestor operații.

9.8 Încărcarea programului la orice adresă

Programarea acceleratorului este de obicei făcută la începutul calculului, programele fiind inițial încărcate pornind de la adresa 0. Pentru a îmbunătății modul de utilizare și programabilitatea acceleratorului, au fost făcute modificări astfel încât programul să poată fi salvat începând cu orice locație din memoria de instrucțiuni.

9.9 Operații cu registrul de adresare

Pentru o și mai eficientă folosire a registrului de adresare (în afară de ce ce a fost descris în Capitolul 8.5), au fost adăugate operații suplimentare pentru modificarea valorii sale. Aceste operații sunt adunări între valoarea curenta a registrului și fie acumulatorul, fie biții de "valoare" din instrucțiunea curentă.

Aceste instrucțiuni au permis o și mai mare flexibilitate pentru controlul adresării, permițând "offset" calculat în acumulator sau din instrucțiune să fie adăugat direct la registrul de adresare fără instrucțiuni suplimentare de încărcare sau descărcare.

Capitolul 10
Rezultatele sintezei

10.1 Rezultate

Acest capitol prezintă rezultatele sintezei pentru diferite configurații ale mașinii de calcul, cele mai importante fiind cele prezentate in Tabelul 10.1 și în Tabelul 10.2. Sinteza condițională este folosită, în funcție de funcționalitatea dorită se sintetizează sau nu anumite blocuri hardware. Sinteza a fost efectuată pentru un Zynq 7020 SoC, mai specific un Pynq-Z2, folosind Vivado 2022.2.2.

Parametrii suplimentari care nu sunt parte din acest studiu sunt:

- Dimensiunea memoriilor: memoria de date: 1024 cuvinte, memoria de instrucțiuni: 2048 cuvinte, memoriile pentru intrare, ieșire și program: 1024 cuvinte;
- "Controller-ul" este sintetizat complet;
- Decodorul pentru instrucțiunile celulelor este plasat în "Controller";
- Dimensiunea datelor: 32 biți.

Tabelul 10.1 Rezultatele sintezei: Map-Reducere

Nr. celule	Slice LUT	Slice REG	F7+F8 Mux
8	9675	3827	761
16	11997	5337	765
32	17012	8654	752
64	26710	15152	743
128	46889	28031	734

Tabelul 10.2 Rezultatele sintezei: Map-Prefixe-Permutare-Grupare, celule maximale

Nr. celule	Slice LUT	Slice REG	F7+F8 Mux
8	17683	6662	843
16	28512	11837	918
32	50957	23959	1024
64	97481	50703	1258
128	206631	110611	1772

10.2 Concluzii

Din sinteze, putem trage următoarele concluzii:

- LUT-urile sunt principala resursă ce este consumată cu creșterea acceleratorului.
- Numărul de BRAM-uri și de DSP-uri disponibile pe fiecare placă, de asemenea limitează numărul de celule de calcul ce poate fi generat. Aceasta nu este o restricție dură deoarece pentru FPGA-uri de vârf DSP-urile sunt în număr de câteva mii, si BRAM-urile cel puțin câteva sute. RAM se poate obține și din LUT-uri, ca Ram distribuit. Blocuri de tip UltraRAM sunt și ele o opțiune
- Dimensiuni de 256-1024 de celule pot fi sintetizate pe FPGA-uri moderne de dimensiuni mare, numărul de celule depinzând de particularitățile arhitecturii. Mecanismul de parametrizare și reconfigurare descris în Capitolul 4 se dovedește foarte valoros pentru optimizarea arhitecturii pentru anumite clase de probleme (ce nu necesită unele module fizice).
- Noul mecanism de IO, modificările din "Controller" și modulele necesare la nivel de sistem ocupă aproximativ 11 mii de LUT-uri. Acesta este un număr relativ mare pentru cazul cu puține celule, dar devine din ce în ce mai puțin relevant cu cât crește numărul de celule.
- Din punct de vedere al folosirii LUT-urilor, o celulă maximală este de aproximativ cinci ori mai mare decât o celulă minimală.
- Suportul hardware pentru calcul în virgulă mobilă consumă foarte multe resurse și ar trebui evitat, dacă aplicația permite asta. Calcule în virgula fixă sunt preferate.
- Suportul hardware pentru calcule în virgulă mobilă necesită optimizări majore privind dimensiunea.
- Rețeaua multifuncțională ocupă o cantitate relativ mare de spațiu. Dimensiunea sa este de $n \times (2 \times \log_2 n - 1)$ celule, (aproximativ de patru ori numărul de celule din rețeaua de reducere.

Descrierea curentă este un amestec de descriere structurală și comportamentală și obiectivul ei principal este de a valida arhitectura. În următoarele etape de dezvoltare, circuitul va fi optimizat pentru a-i reduce dimensiunile.

Capitolul 11
Îmbunătățiri la nivel de aplicație: Algoritmul AES

11.1 Introducere

Acest capitol detaliază implementarea algoritmului de criptare AES pe arhitectura de tip Map-Prefixe-Permutare-Reducere dezvoltată. Această cercetare a fost publicată în [20].

AES-ul este un cifru de tip bloc ce a fost ales guvernul SUA pentru a urma standardului DES. AES-ul este un cifru bloc cu cheie simetrică ce suportă multiple dimensiuni ale cheii: 128, 192, 256 biți. AES-ul suporta multiple variante de codare, precum : ECB), CBC, CTR și altele [21].

11.2 Implementare și rezultate

Algoritmul AES are două etape: generarea de cheie și rundele de calcul. Generarea cheii este folosită pentru a transforma cheia inițială în multiple chei ce vor fi folosite în rundele de codare. Cheia poate avea 128, 192 sau 256 biți, dimensiunea cheii dictează numărul de runde al algoritmului.

În lucrarea noastră, am implementat AES-ul într-o abordare pur SIMD, fiecare celulă de calcul procesând un bloc de text folosind cheia calculată și memorată în "Controller".

Prima etapă a algoritmului (expandarea cheii) se face doar in Controller pentru a reduce consumul energetic. În următoarele etape "Controller-ul" este folosit atât pentru a gestiona controlul buclelor cât și pentru precalcularea adreselor de memorie și pentru distribuirea cheii în toate celulele. În celule, datele sunt stocate ca 8 biți per locație de memorie. Etapa de substituție a bytes-ilor folosește un tabel de tip "look-up" stocat în fiecare celula. Etapele de shift-are a rândurilor și mixare a coloanelor rearanjează datele. Operația de adunare cu cheia este un xor între byte-ul de date calculat și cheia primită.

Algoritmul AES a fost implementat pentru toate cele trei dimensiuni ale cheii și pentru două moduri de operare: ECB (modul de bază) și CTR (modul cu numărare). Rezultate privind latența acestor implementări se pot vedea în Tabelul 11.1. Etapa de expandare a cheii adaugă 5459 (cheie pe 128 biți), 5741 (cheie pe 192 biți), 8106 (cheie pe 256 biți) cicli de ceas suplimentari.

Timpul de transfer poate fi negat prin mecanismele descrise în [13] și [15].

Tabelul 11.1 AES 128 ECB - rezultatele latenței în cicli de ceas[20]

	Număr de celule			
Număr de blocuri	16	32	64	128
16	9353	n/a	n/a	n/a
32	18942	9353	n/a	n/a
64	37688	19086	9353	n/a
128	75180	37832	19358	9353
256	150164	75324	38104	19886
512	300132	150308	75596	38632

Conform cu [20] "AES192 urmează aceleași tendințe, având o creștere de aprx 21% fața de AES128. Similar, AES256 are o creștere a timpului de execuție de 42% comparat cu AES128. Aceleași tendințe apar si la varianta ECB și la varianta CTR.

Comparat cu varianta ECB, varianta CTR crește cu aproximativ 1.5%, 1.3% și 1%, în funcție de dimensiunea cheii. Acest lucru se poate observa pentru toate combinațiile de număr de blocuri și număr de celule de calcul. Modul CTR adaugă un număr redus de instrucțiuni la finalul calculului din modul ECB."

Din punct de vedere al debitului per celulă de calcul, dacă toate sistemele de calcul comparate sunt scalate la aceeași frecvență, acceleratorul propus se situează mai jos decât alte soluții de pe piață, din cauza unității de calcul foarte simple. Suntem totuși în același ordin de mărime ca alte soluții, obținând 2.7 MB/s per core, daca scalăm frecvența la 1.6 GHz. Implementarea GPU din [22] atinge 5.2 MB/s per core iar implementările CPU moderne ating peste 10 MB/s per core la respectiva frecvență.

Un al doilea aspect important e puterea consumată. Datorită simplității elementelor de calcul și bazându-ne pe estimări anterioare de putere, ar trebui sa putem obține aproximativ 2x Gbps/W comparat cu soluții de tip GPU.

11.3 Concluzii

Algoritmul AES a fost implementat pe acceleratorul nostru iar implementarea aceasta oferă performanțe decente, dar nu foarte bune din punct de vedere al debitului și al latenței. Implementarea curentă a fost făcută având o abordare pur SIMD.

Din cauza naturii algoritmului AES, resursele hardware ale acceleratorului nu sunt folosite la maxim, mai specific, rețeaua multifuncțională nu este folosită. Comparat cu o mașina cu mono-nucleu, accelerarea este obținută din separarea calculului între "Controller" și celule și din existența a numeroase unități de calcul.

Capitolul 12
Îmbunătățiri la nivel de aplicație: transpusa unei matrice pătrate

12.1 Introducere

Transpusa este una din cele mai de bază operații de mutare a datelor din algebra lineara. Ea presupune schimbarea liniilor și a coloanelor unei matrice astfel încât linia0 să devina coloana0 și așa mai departe. Pe o mașină pur serială, algoritmul de transpusă este în O(n^2), trecând secvențial prin toate elementele din matrice.

12.2 Abordarea Map-Permutare

Abordarea noastră este structurată în jurul funcției de permutare din rețeaua multifuncțională. Folosind această rețea, un vector de dimensiune "n" poate fi transpus deodată. Astfel se reduce complexitatea temporala a algoritmului la O(n). În interiorul "Array-ului", datele sunt stocate câte una per locație de memorie din celulă, așa cum se poate observa în Figura 12.1 pentru o matrice 16×16 pe o mașină cu 16 celule. Termenul de "vector vertical" va fi folosit pentru coloane și termenul de "vector orizontal" pentru linii.

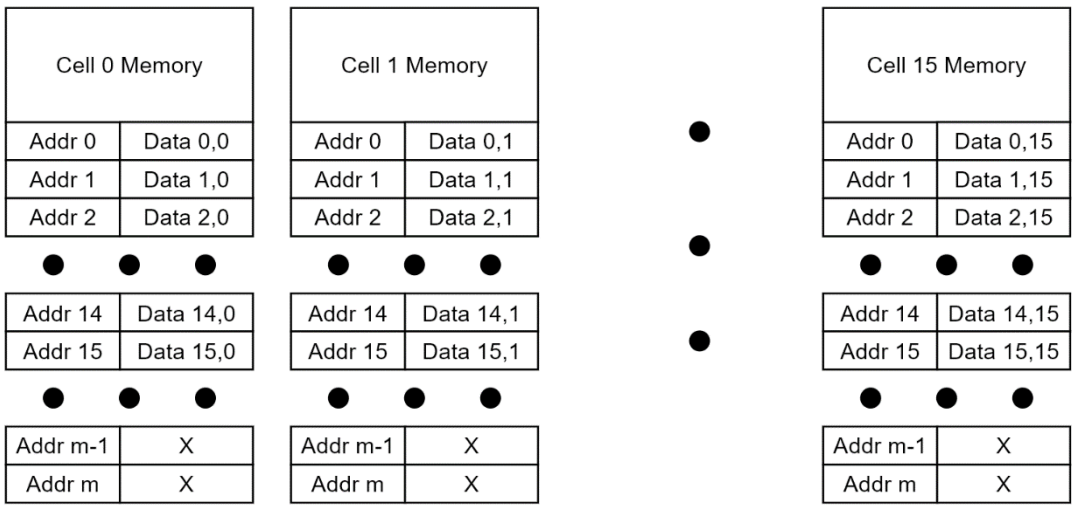

Figura 12.1 Aranjarea datelor din "Array" pentru o matrice de 16x16 pe 16 celule

Abordarea noastră se bazează pe diagonale întrerupte glisante, mai specific, diagonale principale. Datele dintr-o diagonala întreruptă sunt procesate în același timp, în total "n" valori. La fiecare iterație a buclei principale, fiecare celulă va scoate și va primi câte o valoare. Pentru un algoritm de transpusă ce procesează câte "n" valori deodată, problema principală ce trebuie rezolvată este evitarea coliziunilor, atât la citire cât și la scriere. O celulă poate să scoată și să primească doar o valoare per iterație. Din acest motiv, diagonalele întrerupte au fost alese. Acest tipar de acces se potrivește perfect cerințelor mai sus menționate, dar va necesita niște calcul suplimentar pentru pregătirea adreselor ce vor permite scrierea și citirea din locațiile corecte. Algoritmul are 3 etape și este exemplificat în Tabelul 12.1:

- Încărcarea unei diagonale întrerupte;
- Permutarea;
- Salvarea unei diagonale întrerupte.

Tabelul 12.1 Exemplu de transpusă pe arhitectura Map-Permutare

	pasul 1	0 5 a f	4 9 e 3	8 d 2 7	c 1 6 b
Matricea inițială:	pasul 2	0 5 a f	3 4 9 e	2 7 8 d	1 6 b c
0 1 2 3 4 5 6 7 8 9 a b c d e f	pasul 3	0 x x x x 5 x x x x a x x x x f	0 4 x x x 5 9 x x x a e 3 x x f	0 4 8 x x 5 9 d 2 x a e 3 7 x f	0 4 8 c 1 5 9 d 2 6 a e 3 7 b f

Folosind această abordare, biții de permutare trebuie să fie precalculați, salvați în memoria celulelor și retrași la momentul potrivit. Optimizări au fost făcute asupra versiunii inițiale a algoritmului ceea ce a dus la o complexitate în timp din clasa O(n) cu o constantă de aproximativ 13.

În cazul în care se vrea transpunerea unei matrice ce nu este pătrată, trebuie adăugate zerouri până se ajunge la o matrice pătrată.

Capitolul 13
Îmbunătățiri la nivel de aplicație: Transformata Fourier Rapidă (FFT)

13.1 Introducere și pași

Acest capitol detaliază proiectarea, funcționarea și performanțele obținute pentru algoritmul de transformată Fourier rapidă (FFT) pe arhitectura noastră de tip Map-Prefixe-Permutare. Acestea au fost prezentate și în [23].

Considerat unul din algoritmii fundamentali ai computației moderne, algoritmul FFT este folosit de fiecare dată când se dorește calculul transformatei Fourier discrete și un algoritm bazat pe definiția acesteia ar fi prea lent. Algoritmul reduce complexitatea din $O(n^2)$ în $O(n \times \log_2(n))$.

Cea mai cunoscută varianta de FFT este cea bazată pe algoritmul Cooley-Tukey, bazat pe calculul recursiv al unor transformate din ce în ce mai mici [24].

Deoarece se folosesc numere complexe, partea reală și partea imaginară au fost salvate în două locații succesive de memorie, iar operațiile în sine au necesitat mai mulți pași. Adunarea complexă necesită două adunări iar înmulțirea complexă, patru înmulțiri, o adunare și o scădere. Pe arhitectura noastră s-au folosit numere complexe în virgulă fixă, format 8.8, deoarece unitatea de calcul în virgula mobilă nu fusese încă scrisă. Multiplicarea în sine, fiind în virgula fixă 8.8, necesită doua operații: o înmulțire și o shift-are la dreapta cu 8.

13.2 FFT starea artei

Fiind un algoritm foarte important, FFT-ul a fost implementat pe toate platformele folosite în ziua de azi: GPU, FPGA, DSP.

Studii privind implementarea FFT pot fi găsite la [25] și [26].

Implementări anterioare ale FFT-ului pe arhitectura noastră (versiune mai veche) se pot găsi la [27] și [28].

13.3 FFT implementare și variante

Cum se poate vedea in Capitolul 13.1, FFT-ul are nevoie de două tipuri de operații:

- Mutarea datelor;
- Computație propriu-zisă: adunare, înmulțire.

În adaptarea acestui algoritm pentru arhitectura noastră, cele două categorii de operații au trebuit avute în vedere. Computația se efectuează în celulele de calcul, iar deplasarea datelor prin rețeaua multifuncțională, mai specific, folosind funcția de permutare.

Toate calculele au fost făcute în virgulă fixă 8.8 deoarece unitatea de calcul în virgulă mobilă nu fusese încă dezvoltată. Comutarea către virgulă mobilă ar trebui sa fie relativ simplă, performanțele vor avea totuși de suferit, operațiile în virgulă mobilă necesită aproximativ șase cicli de ceas (a se vedea Capitolul 7).

Validitatea algoritmului a fost testată prin comparația cu o implementare în C++.

Pentru a testa hardware-ul, multiple variante de algoritm au fost proiectate, pentru dimensiuni mai mari sau mai mici ale FFT-ului și pentru moduri diferite de a organiza datele. Organizarea Verticală se referă la datele ținute într-o singură celulă de calcul, la multiple adrese de memorie, iar organizarea Orizontală se referă la datele ținute în multiple celule, la aceeași adresă de memorie.

Variantele sunt următoarele:

- Pur serială - o implementare pur serială pentru o mașina mono-nucleu
- Variantă "Dual core" - "Controller" + 1 celulă de calcul
- Variantă "Many-core": Verticală. Este generalizarea la multe nuclee a variantei precedente. Este alcătuită din "Controller" și oricât de multe celule de calcul se dorește. FFT-urile calculate în fiecare celulă sunt complet independente una de alta.
- Varianta "Many-core": Orizontală. Această variantă folosește toate celulele și paralelismul prezent în arhitectura noastră pentru a reduce timpul de execuție al unei transformate. Operațiile sunt efectuate în fiecare celulă iar mutarea datelor folosind rețeaua de permutare. Această organizare oferă cea mai mică latență din toate variantele studiate. Ea are totuși un debit mai mic decât varianta Verticală. Există o scădere logaritmică a performanțelor, ce provine din latența introdusă de rețeaua multifuncțională.
- Varianta "Many-core": Dreptunghi. Versiunea dreptunghiulară este utilă în momentul în care numărul de eșantioane este mai mare decât numărul de celule. Datele sunt organizate similar cu versiunea orizontală, doar că acum vor ocupa multiple locații de memorie. În această situație o combinație dintre versiunea verticală și cea orizontală este folosită.
- Varianta "Many-core": Pătrat. Suplimentar față de versiunea anterioară, dacă numărul de celule este egal cu numărul de linii de memorie ocupate, un algoritm diferit, mai eficient, poate fi folosit.
- Organizarea cu multiple dreptunghiuri/pătrate mici. Această organizare permite un control fin al compromisului dintre latență și debit. Folosirea a mai puține celule pentru calculul unui FFT crește debitul, dar și latența, pe când folosirea a mai multe celule per FFT le va crește pe amândouă.

13.4 Rezultatele simulării FFT

În această secțiune, rezultatele a multiple variante ale algoritmului FFT pe arhitectura noastră sunt prezentate. Toate simulările au fost făcute având ca țintă un sistem SoC ZYNQ7020. Accelerarea debitului și a latenței sunt calculate având ca referință o mașina secvențială, mono-nucleu, cu un set de instrucțiuni asemănător. Rezultatele sunt exprimate în cicli de ceas pentru a evidenția îmbunătățirile arhitecturale și nu cele tehnologice. Operațiile de transfer nu sunt incluse deoarece la momentul testării, acea parte a sistemului încă nu era complet funcțională. Timpul de transfer poate fi negat, așa cum este prezentat în [13]. Rezultatele evaluării sunt luate fie direct din simulări fie din estimării pe baza simulărilor și a metricilor de cod ASM.

Rezultatele discutate în acest capitol pot fi sumarizate în Tabelul 13.1 ce arată un FFT având 256 eșantioane în multiple configurații pe o mașină cu 64 de celule. Acesta au fost făcute pentru a evidenția versatilitatea arhitecturii noastre și compromisul dintre debit și latență.

Tabelul 13.1 Rezultate FFT: Multiple versiuni cu 256 eșantioane

Număr celule	Eșantioane per FFT	FFT-uri în paralel	Cicli per eșantion	Acc. debit	Acc latență	Latență în cicli de ceas	Organizare
64	256	64	4.2	95.9	1.5	69752	Vertical
256	256	1	3.5	114.0	114.0	908	Orizontal
64	256	1	12.1	33.1	33.1	3121	Dreptunghi
64	256	2	10.7	37.5	18.7	5508	Dreptunghi
64	256	4	9.4	42.8	10.7	9659	Dreptunghi
64	256	8	8.1	49.5	6.1	16702	Dreptunghi
64	256	16	6.8	58.6	3.6	28257	Dreptunghi
64	256	4	4.5	89.2	22.3	4640	Pătrat

13.5 Comparație cu alte sisteme

"Pentru comparație, FFT 4096 a fost testat pe trei alte sisteme: testare locală (Intel I7770HQ, 8CPU, 2.8 GHz) a unei implementări CPU a algoritmului FFTW [29], testare locală (Nvidia GeForce GTX1050, 640 core-uri, 1.354 GHz) a unei implementări GPU [30] și implementarea unui IP Xilinx pentru FPGA (pentru ZYNQ 7020 SoC, setări nemodificate din Vivado) [31].

Scalate din punct de vedere al frecvenței, implementarea CPU atinge aproximativ 126 Mega eșantioane pe secundă, implementarea GPU 175, și acceleratorul nostru de uz general 207. O mențiune importantă este că implementarea noastră este în virgulă fixă, ceea ce ar produce o scădere a performanței de aproximativ 3x când se face trecerea către virgulă mobilă (cum sunt implementările CPU și GPU). O altă mențiune importantă este că mașina noastră folosește 64 de nuclee pe când GPU-ul 640, mașina noastră fiind mult mai eficientă atât arhitectural cât și energetic. Folosind configuratorul de IP Xilinx (setări nemodificate, pentru FFT în virgulă fixă) se obține o latență de 14465 cicli, aproximativ jumătate din ce se obține pe arhitectura noastră/ Totuși acest

IP poate efectua doar calcul de transformată Fourier, spre deosebire de arhitectura noastră de uz general."[23]

Comparațiile cu alte acceleratoare găsite în literatură ne plasează între soluțiile GPU și cele FPGA. Acest rezultat era de așteptat deoarece soluțiile FPGA implementează circuite specifice pentru problema ce trebuie rezolvată, astfel oferind latențe și arii mai mici, dar plătind în versatilitate.

13.6 Concluzii

Algoritmul FFT, unul din algoritmii fundamentali din domeniul procesării de semnal, se pretează foarte bine pentru a fi accelerat pe arhitectura propusă. Această muncă dezvoltă și îmbunătățește lucrările prezentate în [27] și [28].

Tiparele de calcul paralel folosite în acest algoritm sunt "Map" (pentru calcul) și Permutarea, ambele fiind tipare implementate eficient în mașina noastră. Algoritmul are o intensitate operațională ridicată (prin care înțelegem cantitatea de calcul ce se efectuează per cantitate de date transferate) și este puternic paralelizabil, folosind toate sau aproape toate resursele de calcul prezente în arhitectură.

Multiple moduri de organizare a datelor au fost testate. Acestea s-au dovedit eficiente în funcție de criteriul de performanță ales. Organizarea Verticală a datelor este foarte eficientă când se dorește debit ridicat, iar organizarea Orizontală oferă cea mai bună latență. Combinații între cele două organizări controlează numărul de celule atribuite fiecărei transformate în parte și oferă un control fin asupra compromisului dintre debit și latență.

Un mod nou de a calcula FFT-ul a fost proiectat si testat, fiind specific arhitecturii propuse. Prin acesta (organizarea în formă de pătrat) se poate obține un debit ridicat, fără a afecta prea mult latența. În acest algoritm, latența produsă de funcția de permutare este ascunsă prin tehnici de tip "pipeline", lucru care în abordarea dreptunghiulară nu se făcea.

Capitolul 14
Îmbunătățiri diverse

14.1 Studiu: Evitarea latențelor produse de tipare de calcul paralel cu circuite cu adâncime logaritmică

Bazându-se pe algoritmii descriși anterior și pe experiența noastră în scrierea codului ASM pentru accelerator, un studiu a fost făcut legat de tehnici pentru reducerea latențelor din rețele de tip Prefixe-Permutare-Reducere.

Am identificat patru metode prin care latențele pot fi evitate sau ascunse, bazate pe patru aplicații dezvoltate. Acest studiu a fost prezentat și publicat la [32]. Majoritatea metodelor sunt variații asupra conceptului de pipeline. Un alt aspect important discutat este ca în unele situații un algoritm complet nou poate să apară și să îmbunătățească semnificativ performanțele. Algoritmii și soluțiile pentru: înmulțirea matrice-vector (pipeline), transpusa unei matrice pătrate (pipeline pe blocuri), FFT (algoritmi noi) și "pooling" (pipeline la nivel de funcții) sunt discutate.

14.2 Modulul de test: suport pentru programe și IO

Modulul de test original era inflexibil și deci o rescriere a acestuia a trebuit să fie făcută, pentru a permite o mai ușoară procesare a programelor și datelor ce trec prin acesta. Pentru ușurarea procesului de testare, un mecanism de testare comparativă a ieșirii cu un fișier de referință a fost de asemenea făcut.

14.3 Modulul de test: Afișare și suport pentru depanare

Depanarea simulărilor Verilog este un proces repetitiv și dificil. Pentru a simplifica acest proces, un mecanism complex de afișare către consolă și către fișiere a fost dezvoltat. Acesta permite afișarea în mod continuu sau doar la finalul simulării a tuturor resurselor hardware prezente în mașină.

14.4 Expandarea macro-urilor

Deoarece acest proiect este scris foarte parametrizat, diferite arhitecturi pot fi ușor generate modificând minimal un set de parametrii. Suplimentar, modificări ale setului de instrucțiuni pot produce o renumerotarea a tuturor codurilor de operație. Pentru a putea urmării valorile macro-urilor folosite, un program ce le scrie într-un format ușor de citit a fost necesar.

Acest program primește fișiere Verilog/SystemVerilog ce conțin macro-uri la intrare și generează un fișier de ieșire de tip map/dicționar compus din perechi de tipul: "nume_macro valoare_macro".

14.5 Petalinux și sistem

Versiunea precedentă a sistemului cu accelerator folosea o imagine Linux distribuită de TUL, creatorii plăcii PYNQ-Z2. Pentru a avea control total asupra nivelului software am hotărât sa recompilăm imaginea Linux. Xilinx recomandă compilarea folosind Petalinux, un set de unelte software folosite în dezvoltarea de Linux embedded.

Având acces complet la sistemul de operare, mai multe modificări de nivel jos au fost posibile. Cea mai importantă dintre acestea este rezervarea unui spațiu de memorie fizică doar pentru accelerator și aplicațiile care vor să îl folosească, acest spațiu fiind indisponibil altor aplicații.

În ciuda numeroaselor probleme întâlnite, imaginea Linux a fost încărcată pe placa PYNQ-Z2 (rulând pe ARM) și folosind SSH, log-area pe placă este de asemenea posibilă.

Într-o primă etapă, aplicațiile au fost scrise prin Vitis SDK fiind apoi mutate și testate pe placă. O altă îmbunătățire a fost adăugarea unui mediu complet de compilare pe placă. Scrierea aplicațiilor s-a dovedit a fi problematică, deoarece unele "driver-e" nu existau și a trebuit sa fie scrise.

Sistemul cu Petalinux dezvoltat încă rulează și este folosit pentru testarea a multiple versiuni atât ale hardware-ului cât și are kit-ului de dezvoltare al aplicațiilor (SDK).

Capitolul 15
Concluzii

15.1 Obiective și rezultate

Cercetarea mea s-a concentrat pe îmbunătățirea unui sistem de calcul eterogen de tip Map-Reduce. Îmbunătățiri au fost făcute atât pentru hardware cât și pentru software.

Din punct de vedere hardware, trei categorii de îmbunătățiri au apărut:

- lizibilitate, reconfigurabilitate, parametrizare;
- îmbunătățiri mici și depanarea erorilor;
- îmbunătățiri mari ce au necesitat adăugarea de blocuri hardware dedicate.

Fiecare din aceste categorii au primit atenția și timpul cuvenit, astfel încât a rezultat un accelerator paralel de tip Map-Prefixe-Permutare foarte puternic, ce poate fi ușor reconfigurat ca să se potrivească cu particularitățile aplicației ce urmează a fi rulate și cu constrângerile de spațiu.

Îmbunătățiri mari au fost făcute pentru a permite, calcul în virgulă mobilă și operații de tip prefixe și permutări. Introducerea conceptului de stivă în accelerator, poate să ajute cu reducere timpului de execuție al unor bucle critice. Mai multe tipuri de shift-ări și rotiri au fost introduse, atât la nivelul acceleratorului cât și la nivelul celulelor individuale.

Deși unele adăugiri au fost făcute ca un răspuns la o problema sau situație specifică, unele au fost făcute și preventiv, pentru a pregătii arhitectura pentru situații viitoare. Acesta este cazul pentru generalizarea mecanismului de IO, adăugarea unora din instrucțiunile de salt sau mecanismul de DMA implementat în "Controller". De asemenea, numeroase erori au fost găsite și corectate.

Din punct de vedere al aplicațiilor, cod în limbaj de asamblare a fost scris atât pentru a testa noile adăugiri cât și pentru a le folosii și evidența în calcule mai complexe. Algoritmul AES a fost adaptat pentru acceleratorul nostru, un nou algoritm pentru transpusa unei matrice pătrate a fost dezvoltat și apoi folosit pentru a calcula mai eficient transformata Fourier. Algoritmul FFT a fost de asemenea implementat pe arhitectura propusă în multiple forme. În funcție de particularitățile aplicației și de compromisul dintre debit și latență acceptat, se selectează organizarea datelor și ce fel de FFT se va face.

Acest sistem a fost apoi sintetizat și implementat pe o placă PYNQ-Z2 SoC pentru a-i testa funcționalitatea. O imagine Linux a fost compilată și rulată pe un ARM Cortex A9 pentru a putea apoi compila și rula aplicații C/C++.

Pe scurt, un sistem de calcul eterogen complet funcțional a fost proiectat, îmbunătățit și implementat, un sistem ce oferă performanțe foarte bune din punct de vedere al paralelismului, scalabilității, latenței și consumului energetic.

15.2 Publicații originale

1) Mihai Antonescu, Gheorghe M. Stefan, "Politehnica" Univ of Bucharest, Romania: “Multi-function Scan Circuit”, International Semiconductor Conference CAS2020; IEEE conference proceedings; doi: 10.1109/CAS50358.2020.9268048.

2) Mihai Antonescu, Mihaela Malița, Gheorghe M. Ștefan “Latency Hiding of Log-Depth Scan and Reduce Networks in Heterogeneous Embedded Systems”, 29th International Symposium for Design and Technology in Electronic Packaging (SIITME), 2023, IEEE conference proceedings.

3) Mihai Antonescu, Gheorghe M. Ștefan, "Multi-Function Scan Circuit for Assisting the Parallel Computational Map Pattern", lucrare acceptată la "Romanian Journal of Information Science and Technology (ROMJIST)", Vol 27, Nr. 1 of 2024,.

4) Andreea-Cătălina Pietricică, Mihai Antonescu, George-Vlăduț Popescu. "Evaluation of AES Cryptographic Algorithm on a General-Purpose Map-Scan Accelerator", International Semiconductor Conference CAS2023, IEEE conference proceedings, DOI: 10.1109/CAS59036.2023.10303705.

5) Mihai Antonescu, Mihaela Malița, “FFT on a Heterogeneous System with a General-Purpose Map-Scan Accelerator”, lucrare acceptată la "Romanian Journal of Information Science and Technology (ROMJIST)",.

6) M. Antonescu, C. Bîra, “Discrete Gravitational Search Algorithm (DGSA) applied for the Close-Enough Travelling Salesman Problem (TSP / CETSP)”; International Semiconductor Conference CAS2019 Sinaia România; IEEE conference proceedings; DOI:10.1109/SMICND.2019.8923719.

7) M. Vasile, S. Martoiu, N. Boukadida, M. Antonescu, A. Ulmamei, G. Stoicea, R. Hobincu, C. Iordache and on behalf of the ATLAS TDAQ collaboration, “FPGA implementation of RDMA for ATLAS readout with FELIX at high luminosity LHC”, published 20 May 2022 • © 2022 IOP Publishing Ltd and Sissa Medialab, Journal of Instrumentation, Volume 17, May 2022, DOI 10.1088/1748-0221/17/05/C05022.

15.3 Contribuții originale

Contribuțiile mele originale se pot împărții în doua categorii: îmbunătățiri hardware și programe scrise în limbaj de asamblare.

Din punct de vedere hardware, am făcut următoarele:

- Rețeaua de reducere a fost îmbunătățită pentru a permite operații de tip prefix, permutări și grupări (pack);
- Parametrizarea și sinteza condiționată pot fi folosite pentru a adapta acceleratorul la aplicația pe care o va rula;
- Mecanismul de IO a fost îmbunătățit pentru a suporta interfețe externe de diferite dimensiuni și moduri diferite de a grupa celulele de calcul;
- Arhitectura cu acumulator a devenit arhitectură cu stivă;
- Registrul global de shift-are a fost îmbunătățit pentru a permite o mai bună comunicare între celulele vecine;
- Un "Barell shifter" a fost adăugat în fiecare celulă pentru a permite operații de tip rotire și shift-are;
- Îmbunătățit mecanismul de adresare al memoriei, atât în "Controller" cât și în "Array";
- Setul de instrucțiuni a fost îmbunătățit și comprimat;
- Mai multe tipuri de instrucțiuni de salt au fost adăugate pentru un mai bun control al programului;
- Alte instrucțiuni au fost adăugate;
- Descoperit erori și le-am reparat.

Din punct de vedere algoritmic, următorii algoritmi au fost făcuți și testați:

- Algoritm pentru calculul codării AES;
- Algoritm pentru calculul transpusei unei matrice pătrate;
- Algoritm pentru calculul transformatei Fourier rapide.

Modulul de test folosit pentru simularea și testarea acceleratorului a fost rescris total și o metodologie de simulare a fost gândită, astfel încât să permită testarea și depanarea rapidă a algoritmilor.

Mai sus de acceleratorul propriu-zis, acesta a fost integrat într-un sistem de calcul pe o placa PYNQ-Z2. Procesorul ARM de pe sistemul ZYNQ a fost folosit ca gazdă pentru acceleratorul nostru. O distribuție de Petalinux a fost compilată de la zero și adaptată nevoilor acceleratorului nostru.

Din aceste contribuții, publicațiile prezentate în Capitolul 15.2 au fost scrise.

15.4 Dezvoltări viitoare

Cum acest proiect este încă în dezvoltare, cantități mari de muncă mai sunt necesare. Din acestea voi menționa mai jos câteva domenii de interes.

În viitorul apropiat sunt două direcții de dezvoltare:

- Consolidare și îndreptarea eventualelor erori - testarea și depanarea erorilor este un pas absolut necesare pentru a merge mai departe, împreună cu o consolidare a efortului făcut deja, în mod special, scrierea documentației.

- Al doilea exemplu de aplicație dezvoltată complet - primul astfel de exemplu a fost înmulțirea de matrice [13]. În acest moment, stiva software s-a schimbat în totalitate, astfel încât o nouă aplicație completă trebuie dezvoltată și testată. Acest test ar trebui să treacă prin toată stiva software, de la nivelul de aplicație C/C++ a utilizatorului final, până la librăria scrisă în limbaj de asamblare și calculul propriu-zis efectuat pe acceleratorul implementat în FPGA.

În viitorul medium apar următoarele dezvoltări:

- Dezvoltarea de librării de primitive în limbaj de asamblare;
- Dezvoltarea librăriei de funcții C/C++ ce vor chema librăria de primitive în limbaj de asamblare;
- Dezvoltarea de instrumente software: asamblor, "profiler", "tuner", depanator, IDE cu interfață grafică (GUI), mediu de rulare (runtime environment), legătura cu git și integrare continuă, cross-compilare, install-ere etc;
- Suport hardware pentru debug pe placă;
- Optimizări ale circuitelor hardware - reducerea numărului de LUT-uri și regiștri folosite, împreună cu lucru la frecvențe cât mai ridicate;
- Testarea unui accelerator de dimensiuni mai mari (128, 256 etc, celule ce calcul);
- Adăugarea de noi instrucțiuni și suport hardware pentru acestea;
- Testarea corespunzătoare având în vedere multiple configurații de parametrii arhitecturali;
- Dezvoltarea și testarea de aplicații complete.

În viitorul îndepărtat, următoarele sunt de cercetat și dezvoltat:

- Moduri multiple de a conecta acceleratorul la calculatorul gazdă;
- Grupuri de acceleratoare - prin natura și forma acestei structuri, conceptul de Map-Reduce se poate extinde teoretic de la acceleratoare pe un singur chip la multiple chip-uri conectate ce formează la rândul lor un accelerator.

Bibliografie

[1] Nvidia, Tensor core usage tips: https://developer.nvidia.com/blog/optimizing-gpu-performance-tensor-cores/, accessed on 20.05.2023.

[2] W. Sun, A. Li, T. Geng, S. Stuijk and H. Corporaal, "Dissecting Tensor Cores via Microbenchmarks: Latency, Throughput and Numeric Behaviors," in IEEE Transactions on Parallel and Distributed Systems, vol. 34, no. 1, pp. 246-261, 1 Jan. 2023, doi: 10.1109/TPDS.2022.3217824.

[3] Hồ, Khoa & Zhao, Hui & Jog, Adwait & Mohanty, Saraju. (2022). Improving GPU Throughput through Parallel Execution Using Tensor Cores and CUDA Cores.

[4] P. M. Basso, F. F. d. Santos and P. Rech, "Impact of Tensor Cores and Mixed Precision on the Reliability of Matrix Multiplication in GPUs," in IEEE Transactions on Nuclear Science, vol. 67, no. 7, pp. 1560-1565, July 2020, doi: 10.1109/TNS.2020.2977583.

[5] John W. Romein, "The Tensor-Core Correlator", in Astronomy and Astrophysics, Vol. 656, Article A52, December 2021, doi: https://doi.org/10.1051/0004-6361/202141896

[6] Sparsh Mittal and Jeffrey S. Vetter. 2015. A Survey of CPU-GPU Heterogeneous Computing Techniques. ACM Comput. Surv. 47, 4, Article 69 (July 2015), 35 pages. https://doi.org/10.1145/2788396.

[7] Pieter Hijma, Stijn Heldens, Alessio Sclocco, Ben van Werkhoven, and Henri E. Bal. 2023. Optimization Techniques for GPU Programming. ACM Comput. Surv. 55, 11, Article 239 (November 2023), 81 pages. https://doi.org/10.1145/3570638.

[8] A. M. Turing, On computable numbers with an application to the Eintscheidungsproblem. Proceedings of the London Mathematical Society 42, 1936.

[9] S. Kleene, "General Recursive Functions of Natural Numbers," Mathematische Annalen 112:727–742. 1936.

[10] G. M. Ștefan, M. Malița: "Can One-Chip Parallel Computing Be Liberated From Ad Hoc Solutions? A Computation Model Based Approach and Its Implementation", 18th International Conference on Circuits, Systems, Communications and Computers (CSCC 2014), Santorini Island, Greece, July 17-21, 2014, 582-597.

[11] M. Malița, G. M. Ștefan, D. Thiebaut: "Not Multi-, but Many-Core: Designing Integral Parallel Architectures for Embedded Computation", ACM SIGARCH Computer Architecture News, 35 (5)32-38, Dec. 2007. Special issue: ALPS '07 - Advanced low power systems; communication at International Workshop on Advanced Low Power Systems held in conjunction with 21st International Conference on Supercomputing, June 17, 2007 Seattle.

[12] Mihaela Malița, Gheorghe Ștefan: "The Berkeley Motifs and an Integral Parallel Architecture", in ROMJIST, vol. 12, no. 1, 2009, pag. 35-49.

[13] George-Vladuț Popescu, " Architectures and Structures for Heterogeneous Computing - Improvements in Data Transfer for a Heterogeneous Computing System", PhD Thesis, UPB, Bucharest 2023.

[14] PYNQ Z2 overview, [online] Available: https://www.tulembedded.com/fpga/ProductsPYNQ-Z2.html, accessed on 24.07.2023.

[15] George-Vlăduț, Popescu. "Improvements in Data Transfer for a MapReduce Accelerator". In: Romanian Journal of Information Science and Technology (ROMJIST), 25.3-4 (2022), pp. 368–380. ISSN: 1453-8245.

[16] Mihai Antonescu, Gheorghe M. Stefan, "Politehnica" Univ of Bucharest, Romania: Multi-function Scan Circuit. CAS2020; International Semiconductor Conference IEEE conference proceedings; doi: 10.1109/CAS50358.2020.9268048.

[17] Mihai Antonescu, Gheorghe M. Stefan, "Multi-Function Scan Circuit for Assisting the Parallel Computational Map Pattern", lucrare acceptată la "Romanian Journal of Information Science and Technology (ROMJIST)", Vol 27, Nr. 1 of 2024.

[18] Vaclav E. Benes: Mathematical Theory of Connecting Networks and Telephone Traffic. New York: Academic Press, 1965.

[19] Abraham Waksman. 1968. A Permutation Network. J. ACM 15, 1 (Jan. 1968), 159–163. https://doi.org/10.1145/321439.321449.

[20] Andreea-Cătălina Pietricică, Mihai Antonescu, George-Vlăduț Popescu. "Evaluation of AES Cryptographic Algorithm on a General-Purpose Map-Scan Accelerator", International Semiconductor Conference CAS2023, IEEE conference proceedings, DOI: 10.1109/CAS59036.2023.10303705.

[21] "Recommendation for Block Cipher Modes of Operation" (PDF). NIST.gov. NIST. p.9. Archived (PDF) from the original on 29 March 2017, [online] Accessed at: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf, on 29.07.2023.

[22] J. Ma, X. Chen, R. Xu and J. Shi, "Implementation and Evaluation of Different Parallel Designs of AES Using CUDA," 2017 IEEE Second International Conference on Data Science in Cyberspace (DSC), Shenzhen, China, 2017, pp. 606-614, doi: 10.1109/DSC.2017.19.

[23] Mihai Antonescu, Mihaela Malița, "FFT on a Heterogeneous System with a General-Purpose Map-Scan Accelerator", lucrare acceptată la "Romanian Journal of Information Science and Technology (ROMJIST)".

[24] Cooley, James W.; Tukey, John W. (1965). "An algorithm for the machine calculation of complex Fourier series". Mathematics of Computation, 1965, Vol 19 (no. 90): 297–301. doi:10.2307/2003354. JSTOR 2003354.

[25] Mario Garrido "A Survey on Pipelined FFT Hardware Architectures", Journal of Signal Processing Systems, vol 94, Jul. 2021.

[26] Konguvel Elango and Mu Kannan "A Survey on FFT/IFFT Processors for Next Generation Telecommunication Systems",Journal of Circuits, Systems and Computers, vol. 27, 03.2018.

[27] I. Lörentz, M. Malița, R. Andonie, "Fitting FFT onto an energy efficient massively parallel architecture" Proceedings of the Second 617 International Forum on NextGeneration Multicore/Manycore Technologies, IFMT '10, 8:1–8:11, 2010.

[28] Calin Bira, Liviu Gugu, Mihaela Malita, Gheorghe Stefan, "Maximizing the SIMD Behaviour in SPMD Engines", Proceedings of 619 the WCECS 2013, Oct 2013.

[29] M. Frigo and S. G. Johnson, "The Design and Implementation of FFTW3," in Proceedings of the IEEE, vol. 93, no. 2, pp. 216-231, Feb. 2005, doi: 10.1109/JPROC.2004.840301.

[30] Alexander Hurd, 2018, "fftw-cufftw-benchmark", Available: https://github.com/hurdad/fftw-cufftw-benchmark, Commit cfc8aa8, Accessed on 08.05.2023.

[31] AMD-Xilinx, PG109 (04.05.2022): Fast Fourier Transform v9.1 LogiCORE IP Product Guide, [online] Available at: https://docs.xilinx.com/r/en-US/pg109-xfft/Fast-Fourier-Transform-v9.1-LogiCORE-IP-Product-Guide, Accessed on 08.05.2023.

[32] Mihai Antonescu, Mihaela Malița, Gheorghe M. Ștefan "Latency Hiding of Log-Depth Scan and Reduce Networks in Heterogeneous Embedded Systems", 29th International Symposium for Design and Technology in Electronic Packaging (SIITME), 2023, IEEE conference proceedings.