# Compilare Portabilă pentru Acceleratorul Vectorial Connex-S pentru Sisteme Îmbarcate (Rezumatul tezei de doctorat)

Autor: Alexandru Emilian Şuşu

În aceste zile experimentăm un moment fertil pentru calculul paralel, în special datorat avansului în tehnologia compilării, maturarea formalismelor matematice pentru aceasta şi popularizarea proiectării hardware.

Contribuţia cea mai importantă a acestei teze de doctorat este compilarea programelor C secvenţiale pentru Connex-S, un accelerator lat competitiv, scalabil şi customizabil, pentru aplicaţii îmbarcate intensive cu 32 până la 4096 unităţi de execuţie (în engleză, *lanes*) cu 16 biţi întregi şi o capacitate limitată locală de memorie de tip "scratchpad".

Acceleratorul Connex-S îşi obţine eficienţa din delegarea compilatorului pentru a extrage paralelism prin vectorizarea codului scalar, şi prin generarea explicită a comunicării într-un mod optimizat între lane-urile procesorului. Evident extragerea paralelismului în software este considerabil mai bună decât atunci când este implementată în hardware, cum este spre exemplu, cazul procesoarelor superscalare.

Connex-S este un procesor low-power. Motivul este că arhitecturile *Single Instruction Multiple Data (SIMD)* de tip array sau vectoriale, pe lângă faptul că sunt un mod scalabil de a proiecta arhitecturi paralele [Kozyrakis & Patterson, 2003], ating o eficienţă energetică mai bună deoarece: (i) procesoarele SIMD (sau array) folosesc considerabil mai puţine resurse pentru unitatea de control decât pentru datapath [Waeijen et al., 2015; Şuşu, 2019; Patterson & Hennessy, 2017], (ii) delegăm compilatorul să extragă *Instruction Level Parallelism (ILP)* în locul hardware-ului aşa cum se întâmplă în procesoarele superscalare, şi (iii) putem menţine puterea computaţională prin scăderea frecvenţei şi a voltajului sursei deoarece putem creşte nivelul paralelismului hardware [Chandrakasan et al., 1992]. [Kozyrakis, 2002] argumentează că într-un procesor SIMD (sau de tip *array*) creşterea numărului de lane-uri obţine o reducere proporţională a consumului de putere fără schimbarea puterii computaţionale maxime a sistemului.

Numărul de lane-uri ale unui procesor SIMD se numeşte *lăţime* (sau *lungime vectorială*).

ISA-ul procesorului Connex-S este descris în teza doctorală. Dar există şi un document tehnic complet [Ştefan, 2015] cu ISA-ul procesorului Connex-S .

Orice procesor digital are nevoie de unelte de dezvoltare bune software care să fie folosite de comunitate. Aceste unelte sunt asamblorul, compilatorul pentru C sau C++, biblioteca de runtime, şi

biblioteci software adiționale. Ideal, biblioteca de runtime conține puține elemente scrise manual în limbaj de asamblare de programatori experți cum ar fi emulare pentru floating point și apeluri speciale de Sistem de Operare. Pe de altă parte, un compilator de C include toată logica complexă să translateze cod general de C către limbaj de asamblare. Chiar dacă aceasta este ceva ușor de făcut pentru procesoare scalare, auto-vectorizarea, compilarea pentru procesoare vectoriale rămâne, chiar și după aproape 40 de ani de cercetare, o sarcină dificilă. Spre exemplu, mult cod neregulat cum ar fi cod cu structuri de date complexe, cod recursiv sau pattern-uri paralele cum ar fi *reducere* sau *scan*, rămâne dificil a fi auto-vectorizat.

Țintele cercetării mele au fost de la început: (i) să creeze un compilator care optimizează, pe cât de general posibil pentru procesorul vectorial Connex-S, împreună cu un bun suport de bibliotecă de runtime; (ii) să propun optimizări de putere și energie pe cât posibil în cadrul compilatorului. Am fost în stare să adresez optimizări de energie numai ca parte a bibliotecii de runtime, dar nu în compilator deoarece infrastructura de management a puterii a procesorului a fost dificil să o construim și sunt puține, optimizări complexe de reducere a consumului de energie care se pot adresa de către compilator.

Deoarece Connex-S este un accelerator slab integrat cu CPU-ul gazda folosește un model de programare similar cu limbajul OpenCL care se numește OPINCAA [Bîră et al., 2013]. De aceea, compilatorul nostru folosește framework-ul LLVM și generează cod OPINCAA, care este mai exact un asamblor vectorial *Just-In Time (JIT)* și bibliotecă de coordonare C++ pentru Connex-S accelerând computații pentru un CPU arbitrar. De aceea, adresăm în middle-end-ul compilatorului aspecte eficiente de vectorizare, comunicare și sincronizare. Efectuăm analize statice cantitative de program folositoare, printre altele, pentru alocatorul de memorie al compilatorului care folosește mărimi simbolice și mecanismul de coordonare al lui OPINCAA. Prin folosirea unui asamblor vectorial JIT și prin encodarea lungimii vectoriale a lui Connex-S ca un parametru în programul generat OPINCAA, asigurăm agnosticism de tip vector-length să suporte execuția dispozitive îmbarcate distinct, cum ar fi câteva camere cu rezoluții diferite, fiecare echipat cu acceleratoare Connex-S de mărime specializată menite să salveze energia pentru kerneluri de procesare de imagine.

Încurajăm a folosi un compilator *Vector Length Agnostic (VLA)*, în principal deoarece sistemele îmbarcate ar trebui proiectate cu unul sau mai multe procesoare Connex-S de mărime specializată adaptate la mărimile de intrare de program frecvente pentru a reduce consumul de energie când accelerăm fiecare din următoarele benchmark-uri: (i) *Basic Linear Algebra Subprograms (BLAS)* cum ar fi multiplicări de matrice, (ii) transformări de *Computer Vision (CV)* sau (iii) *Deep Neural Networks* și alte sarcini intense de machine learning. Experimentele din Figura 14 din [Waeijen et al., 2015] sugerează că pentru a salva energie, noi trebuie să accelerăm acest tip de kerneluri de cod regular cu

procesoare Connex-S cu o lungime vectorială apropiată de cea a numărului de iteraţii a unei bucle vectorizate. De aceea, în caz că producem câteva camere digitale cu diferite rezoluţii (maxime) stabilite din fabrică, trebuind să facă diverse computaţii de procesare de imagine cum ar fi scoaterea zgomotului din imagini prin folosirea unei transformări CV de tip convoluţie pe cadrele capturate de mărimea de fabrică corespunzătoare, pentru a salva energie noi trebuie să folosim un accelerator Connex-S cu lungime vectorială aproape de lăţimea imaginilor.

# 1 Descrierea detaliată a compilatorului pentru Connex-S

Cei mai complecşi paşi ai compilatorului sunt: efectuarea de data tiling, vectorizarea de bucle [Naishlos, 2004; Nuzman&Henderson, 2006] cu analiză statică cantitativă simbolică, şi emulare eficientă nenativă a operaţiilor aritmetice şi logice în back-end.

Prezentăm în Figura 1 etapele compilatorului împreună cu detaliile de cum să folosim programul C++ rezultat pentru diverse platforme îmbarcate cu acceleratoare Connex-S cu lăţime specializată. În primul rând determinăm pentru codul C al nostru care este mărimea kernelului vectorial compilat de la programul original C printr-o simplă JIT-asamblare cu OPINCAA, mărimea memoriei folosite de kernelul vectorial, şi numărul de iteraţii al buclelor imbricate. Obţinem mărimea memoriei folosite cu analiză statică LLVM denumită *Symbolic Range Analysis (SRA)* [Mendonça et al., 2017]. Dacă mărimea memoriei folosite de către programul nostru este o constantă mai mică decât mărimea memoriei locale a lui Connex-S, normal de 256 KB, atunci furnizăm pur şi simplu programul original C către compilatorul Connex-S OPINCAA LLVM. Altfel, trebuie să aplicăm mai înainte tiling de bucle, o transformare standard care sparge o buclă în două bucle imbricate, una exterioară care se cheamă *buclă de tile* şi una internă care se numeşte *buclă de elemente*, cu fiecare buclă de elemente procesând un mai mic set de iteraţii [Kennedy & Allen, 2002]. Într-un astfel de caz, următorul pas este să efectuăm selecţia optimă a mărimii a data tile-ului folosind un model de cost al acceleratorului Connex-S astfel încât să minimizăm timpul de execuţie total al kernelului astfel încât să putem acomoda datele programului mai mari în memoria *Local Storage (LS)* a procesorului. Pentru aceasta folosim PPCG, care ia la intrare program sursă C şi mărimile tile-urile optime. PPCG aplică modelare poliedrală [Pol, 2020] şi analiză de dependenţă de date pentru a verifica dacă tiling-ul de bucle cerut este legal, în care caz programul C transformat este oferit la intrarea compilatorului Connex-S OPINCAA LLVM.

Acum prezentăm paşii efectuaţi în Figura 1 de compilatorul OPINCAA LLVM. Parsăm fişierul sursă C (transformat) cu comanda *clang*, front-end-ul compilatorului, care generează cod LLVM IR neoptimizat. După aceasta comanda *opt* optimizează *IR-ul (Intermediate Representation)* lui LLVM prin execuţia pipeline-ului middle-end-ului LLVM. După aceea, rulăm back-end-ul, llc, pentru a genera instrucţiuni de asamblare pentru CPU şi cod vectorial de asamblare pentru Connex-S. După aceasta

înlocuim buclele vectorizate în fişierul cu codul C sursă cu kernelurile OPINCAA şi cu codul de coordonare asociate generate pentru a obţine program final OPINCAA, prin folosirea unei simple unelte scrise în C++, numit *ReplaceLoopsWithOpincaaKernels*. Cu acest ultim pas, putem considera că am făcut o simplă transformare sursă-la-sursă, care face codul independent de CPU. Generăm cod C++ care foloseşte framework-ul OPINCAA C++. Aceasta permite codului de asamblare Connex-S care este independent de lungime vectorială cu *strip-mining (segmentarea buclei)* făcut în mod parametric după CVL pentru a se baza pe asamblorul OPINCAA JIT, care citeşte de asemenea la runtime valoarea concretă a CVL variabilei de mediu program. Programul C++ OPINCAA rezultat poate fi compilat cu o unealtă standard precum GCC, preferabil la nivelul maxim de optimizare pentru beneficiul codului gazdă.

Deoarece un back-end LLVM trebuie să aibă un procesor scalar CPU, nu doar o unitate vectorială, creăm un back-end Connex-S prin adăugarea instrucţiunilor vectoriale Connex-S pentru LLVM back-end-ul existent *eBPF (extended Berkeley Packet Filter)*. Puteam începe de la un back-end mai popular cum ar fi cel al lui ARM, dar deoarece metoda noastră ignoră codul de asamblare CPU la sfârşit şi îl înlocuieşte cu cod C original nevectorizat, ne permitem să folosim ceva mai simplu. eBPF este un simplu procesor de 64 biţi de tip RISC [The Linux Kernel Archives, 2014], o extensie a arhitecturii BPF [McCanne & Jacobson, 1993], normal interpretat de kernelul Linux (sau alte Unix-uri) şi are cea mai mică implementare de back-end în repository-ul LLVM. Adăugăm la el instrucţiunile vectoriale inspirându-ne din *MSA (MIPS SIMD Architecture)* instrucţiunile vectoriale specificate în back-end-ul Mips.

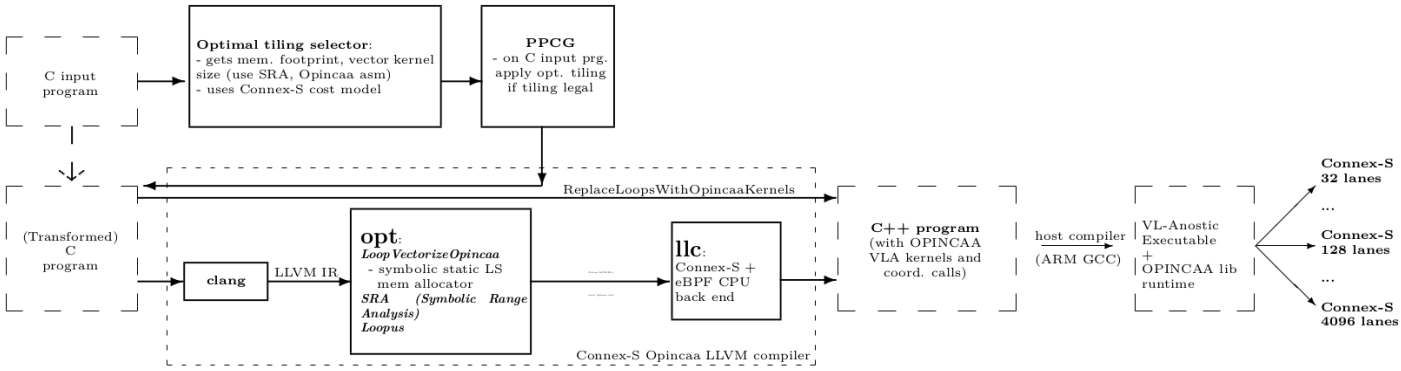


Figura 1: Stagiile compilatorului Connex-S. De asemenea, în partea dreaptă, prezentăm exemple de desfăşurare pe sisteme îmbarcate cu acceleratoare Connex-S de lungime vectorială ajustată (*custom-width*) cu un număr de lane-uri între 32 şi 4096.

În modelul maşină LLVM IR, unitatea vectorială este cuplată strâns la CPU, care rulează codul secvenţial LLVM. Cu toate acestea, Connex-S este un accelerator, slab integrat cu CPU-ul, cu propriul lui spaţiu de memorie separată deci noi trebuie să facem explicit comunicare şi sincronizare dintre cele

două, folosind *API-ul (Application Programming Interface)* de coordonare OPINCAA. De aceea, noi generăm apeluri de coordonare OPINCAA pentru aceste transferuri de memorie OPINCAA pentru transferuri de memorie şi rezultate de reducere, execuţie şi sincronizare în modulul nostru LoopVectorizeOpincaa.

Extindem modulul standard LLVM LoopVectorize, care transformă automat, dacă este fezabil şi profitabil, bucle de nivel maxim interior secvenţiale ale programului LLVM IR în cod vectorial LLVM IR, pentru a implementa cele mai multe noi funcţionalităţi middle-end (noi adăugăm în middle-end de asemenea modulele SRA şi Loopus):

• (i) un alocator de memorie static cu mărimi simbolice pentru memoria *Local Storage (LS)* a lui Connex-S;

• (ii) generarea optimă, agregată a transferurilor I/O dintre RAM-ul sistemului şi memoria Connex-S LS (folosind metodele *writeDataToConnexPartial(), readDataFromConnexPartial()*), al primitivelor de kernel OPINCAA *begin* şi *end*, al apelurile de execuţie şi de citire a rezultatelor de reducere a kernel-ului;

• (iii) logica de a crea noi instrucţiuni vectoriale de citire şi scriere care adresează memoria LS în loc de RAM-ul sistemului;

• (iv) generarea de părţi de început şi sfârşit ale buclelor din kernelul vectorial Connex-S. Avem două alternative să implementăm bucle în kernel-uri vectoriale: folosind bucle C++ *for* în OPINCAA, care de fapt fac *unroll* (în română derulare) total al corpului de instrucţiuni Connex-S sau folosind mecanismul de bucle hardware al lui Connex-S.

• (v) obţinerea liniilor şi coloanelor de început şi sfârşit ale buclelor care se vectorizează din fişierul sursă C pentru a le înlocui mai târziu cu cod OPINCAA. Pentru aceasta, noi folosim clasa *ScopeTree* din compilatorul *DawnCC* [Mendonça et al., 2017].

• (vi) pentru o mai bună performanţă noi avem de asemenea nevoie să reducem mărimea kernel-ului prin generarea pentru cele mai interioare două niveluri de bucle C imbricate a unei bucle OPINCAA REPEAT conţinând o buclă C++ *for* (aceasta normal rezultă într-un kernel cu număr minim de instrucţiuni vectoriale dacă toate numerele de iteraţii originale sunt mai mari sau egale cu CVL), iar pentru bucle C simple generăm numai REPEAT. Această transformare este corectă dacă bucla a doua cea mai interioară practic nu are nici un cod în afară de bucla interioară, în afara celui de iniţializare a variabilei de reducere. Aceasta înseamnă că putem prelucra numai bucle perfect imbricate [Kodukula et al., 1997] cu transformarea noastră. Pentru bucle imbricate mai adânci noi lăsăm pur şi simplu neschimbat buclele exterioare.

Vectorizarea buclei celei mai interioare generează un *basic-block* cu numele *vector.body*, dar de asemenea prolog în blocul *vector.ph* şi epilog în codul vectorial în *middle.block*, în care punem ultima

parte a buclei celei mai interioare, fie o instrucţiune OPINCAA *END_REPEAT* sau o pseudo-instrucţiune de *inline assembly* care să termine bucla C++.

Noi colectăm pur şi simplu tot acest cod vector cu programul *ReplaceLoopsWithOpincaaKernels* şi îl adăugăm în kernel-ul OPINCAA. După cum am discutat mai devreme, pentru eficienţă noi vectorizăm practic de asemenea a doua buclă cea mai interioară dacă nu are nici un cod în afara buclei interne, prin transformarea ei într-o buclă Connex-S *REPEAT*.

Divergenţa de control datorată instrucţiunilor de salt C *if-then-else* este prelucrată de modulul *LoopVectorizeOpincaa*, prin aplicarea transformării de cod *if conversion*, care înseamnă că fluxul de control al programului cu salturi este înlocuit cu instrucţiuni LLVM IR vectoriale *select*. Cu toate acestea, dacă structura buclei care va fi vectorizată este complexă, este posibil ca *if conversion* şi vectorizarea să eşueze în LLVM.

## 2. Automatizarea specificării transformării eficiente a operaţiilor aritmetice şi logice nenative din LLVM IR către cod vectorial de asamblare

Din lipsă de spaţiu, prezentăm pe scurt cele mai interesante rezultate ale metodei semi-automate pe care o folosim pentru emularea eficientă a operaţiilor nenative LLVM IR de tipuri cum ar fi 32 biţi întregi sau 16 biţi floating point folosind compilatorul nostru Connex-S OPINCAA LLVM. Referim cititorul la Capitolul 5.3 din teza doctorală care descrie în mai mult detaliu aspectele acestei secţiuni.

Metoda pe care o propunem consistă din: i) a crea manual kernel-uri OPINCAA cu cod de asamblare vectorial optimizat pe care le numim *bucăţi de cod predefinite* pentru emularea fiecăreia din aceste operaţii nenative; ii) generăm automat cod C++ API pentru modulul LLVM de selecţie a instrucţiunilor, care transformă eficient instrucţiunile nenative către bucăţile de cod predefinite corespunzătoare.

Aceasta permite emularea eficientă a operaţiilor nenative în sensul că compilatorul LLVM generează cod de emulare *inlined* (în cadrul codului) pentru un program cu operaţii nenative aritmetice, optimizate de către toate modulele standard cum ar fi alocarea de regiştri şi *Common Subexpression Elimination (CSE)* practic pornind de la bucăţile de cod manual optimizat.

Pentru ca această metodă să funcţioneze bine am scris un program de generare de cod ca parte a bibliotecii OPINCAA cu un algoritm care automatizează specificarea de cod de API LLVM în C++ pentru modulul de selecţie a instrucţiunilor, deoarece scrierea de cod manual C++ poate să se facă cu greşeli.

# 3. Selectarea optimă a mărimii de tiling a buclelor la compilare

Deoarece Connex-S are o memorie locală scratchpad de capacitate limitată de 256 KB în mod normal,

noi prezentăm cum folosim generatorul de cod PPCG C-to-C pentru a realiza data tiling pentru a minimiza timpul total de execuţie a kernelului, în timp ce punem datele mai largi de program în memorie locală.

Pentru a calcula tiling-ul optim, vom introduce mai întâi un model de cost precis al acceleratorului îmbarcat Connex-S sintetizat în Xilinx Zynq-7020 FPGA, pentru a determina timpul total de execuţie al unei rutine cu tiling. Rulăm la compilare această problemă de optimizare dacă programul sursă C nu are mărime parametrică a array-urilor pentru care nu putem infera costul de transfer static. Munca noastră se aseamănă cu cea a lui [Lin et al., 2010] care construiesc un model de cost similar în principiu cu al nostru pentru a obţine tile optim pentru procesorul IBM Cell.

Connex-S are model de cost precis şi simplu de calculat dacă se întâmplă blocaje puţine în sistem, care se întâmplă pentru kernel-urile care sunt complet vectorizate şi pentru care citim eficient rezultatele de reducere, în blocuri de cel puţin câţiva KB. Aceasta nu este nici o surpriză deoarece Connex-S este un accelerator larg vectorial, cu o *memorie scratchpad (SPM)* controlată din software în loc de memorie de tip cache, care face execuţia vectorială predictibilă, fără nici o execuţie speculativă sau de tip *out-of-order*, şi implementăm Connex-S într-un Xilinx Zynq-7020 FPGA pe Zedboard, o platformă simplă de prototipare de tip îmbarcat. De asemenea, experimentăm blocaje ale execuţiei când FIFO-ul de instrucţiuni şi de reducere devine plin, putând lua sub 5% din timpul total de execuţie al unui kernel pentru tiling-uri nedegenerate. De aceea, realizăm riguros măsurători în multe condiţii diferite pe Zedboard şi model de cost antrenat pentru a avea acurateţe bună.

Furnizăm un exemplu de compilare care necesită a realiza *tiling* de bucle să fie în stare să încapă datele accesate în memoria LS de 256 KB a lui Connex-S.

Când compilăm cu programul PPCG funcţia C simplă din Listing-ul 1, care reprezintă testul *MatMulBT-512* care implementează multiplicarea a două matrice pătrate de 512x512 elemente, cu elemente de tip *int16_t*, stocate în ordine linie după care coloană, cu al doilea operand transpus să permită vectorizarea, obţinem programul din Listing-ul 2.

```
typedef int16_t Type;

#define N 512
Type A[N][N];
Type B[N][N];
Type C[N][N];

void MatMul_BTransposed() {
  int i, j, k;

  for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j) {
      C[i][j] = 0;
      for (k = 0; k < N; ++k)
          C[i][j] += A[i][k] *
                     B[j][k];
    }
}
```

*Listing 1: Programul C sursă pentru multiplicare de matrice*

Acest program C, cu *tiling* de bucle optim astfel încât datele se potrivesc corect în memoria Connex-S LS şi programul să aibă cel mai mic timp de execuţie, este după aceea compilat cu compilatorul Connex-S OPINCAA LLVM.

Prezentăm în Listing-ul 3 programul final conţinând cod de coordonare OPINCAA şi de asamblare vectorial.

Observăm că în Listing-ul 1 nu exprimăm mărimea matricei exprimată ca o variabilă C aceasta făcând problema tractabilă la compilare.

```
int16_t Atile[182][512];
int16_t Btile[74][512];
int16_t Ctile[182][74];

void MatMul_BTransposed() {
  #define min(x, y) (x < y ? x : y)

 #ifdef NOT_FOR_CONNEX_LLVM_COMPILER
  // The i tile loop
  for (int it = 0; it < 512; it += 182) {
    int callWriteDataToConnexForFirstArray = 1;
     // For data reuse in the OPINCAA C++ program

    // Copying data from A into Atile
    for (int ip = 0; ip < min(182, 512-it); ip++)
      for (int k = 0; k < 512; k++)
        Atile[ip][k] = A[it + ip][k];

    // The j tile loop
    for (int jt = 0; jt < 512; jt += 74) {
      // Copying data from B into Btile
      for (int jp = 0; jp < min(74, 512-jt); jp++)
        for (int k = 0; k < 512; k++)
          Btile[jp][k] = B[jt + jp][k];
 #endif
  // Continued on right column
```

```
  // Continued from left column

      //Only this code is vectorized on Connex-S
      for (int ip = 0; ip < min(182, 512-it);
                          ip++) // i point loop
        for (int jp = 0; jp < min(74, 512-jt);
                          jp++) { // j point loop
          Ctile[ip][jp] = 0;
          for (int k = 0; k < 512; k++)
            Ctile[ip][jp] += Atile[ip][k] *
                             Btile[jp][k];
        }

 #ifdef NOT_FOR_CONNEX_LLVM_COMPILER
      int counter = 0;
      // Putting back data from Ctile into C
      for (int ip = 0; ip < min(182, 512-it);
                                    ip++)
        for (int jp=0; jp < min(74, 512-jt);
                                    jp++)
          C[it + ip][jt + jp] = Ctile[ip][jp];

    } // End of jt tile loop
  } // End of it tile loop
 #endif
}
```

*Listing 2: Programul C generat de PPCG din Listing 1, simplificat pentru a fi lizibil, pentru vectorul de mărimi tile (182, 74, 512), pentru o memorie Connex-S LS de 256 KB*

```
int CONNEX_VL;
...
void MatMul_BTransposed() {
  // Assuming: CVL in {32, 64, 128, 256, 512}
  #define min(x, y) (x < y ? x : y)
  // The i tile loop
  for (int it = 0; it < 512; it += 182) {
    int callWriteDataToConnexForFirstArray = 1;
    // Copying data from A into Atile
    for (int ip = 0; ip < min(182, 512-it); ip++)
      for (int k = 0; k < 512; k++)
        Atile[ip][k] = A[it + ip][k];
    // The j tile loop
    for (int jt = 0; jt < 512; jt += 74) {
      // Copying data from B into Btile
      for (int jp = 0; jp < min(74, 512-jt); jp++)
        for (int k = 0; k < 512; k++)
          Btile[jp][k] = B[jt + jp][k];
  // Performing data reuse for block Atile
  if (callWriteDataToConnexForFirstArray == 1) {
    // connexGlobal C++ object encapsulates
    //      accelerator functionality
    connexGlobal->writeDataToConnexPartial(Atile,
                          /* num elems written */
                          min(182, 512-it) * 512,
                        /* LS memory offset */ 0);
    callWriteDataToConnexForFirstArray = 0;
  }
  connexGlobal->writeDataToConnexPartial(Btile,
                        /* num elems written */
                          min(74, 512-jt) * 512,
                  /* LS mem offset */
                  min(182, 512-it) * 512 / CVL);
  // The OPINCAA vector kernel starts here
  BEGIN_KERNEL("allowRedefine_MatMul_BTransposed");
   EXECUTE_IN_ALL(
    R(0) = 0;
    R(1) = 1;
    // The i point loop
    for (int ip = 0; ip < min(182, 512 - it);
                                    ip++) {
 // Continued on right column
```

```
    R(2) = min(182, 512-it) * 512 / CVL;
                  // Btile start offset

    // Translation of j point loop jp
    REPEAT(min(74, 512-jt));
     R(3) = (ip * 512) / CVL;
              // vload index for Atile[ip]
     R(4) = 0; // accumulator

     // Vectorized innermost loop k
     //          (CVL strip-mined dot product)
     for (int kStripmine = 0; kStripmine < 512;
                          kStripmine += CVL) {
                // Read Atile[ip][*] vector
       R(5) = LS[R(3)];
       R(3) += R(1);
                // Read Btile[jp][*] vector
       R(6) = LS[R(2)];
       R(2) += R(1);
       R(6) * R(5); R(5) = MULTLO();
       // Accumulate (for dot prod)
       R(4) += R(5);
     }
     RED R(4); // compute Ctile[ip][jp]
    // End of translation of j point loop jp
    END_REPEAT;
  } // end of i point loop ip
 );
END_KERNEL("allowRedefine_MatMul_BTransposed");

connexGlobal->executeKernel(
            "allowRedefine_MatMul_BTransposed");
connexGlobal->readCorrectReductionResults(Ctile,
              min(182, 512-it) * min(74, 512-jt),
                                sizeof(int16_t));

    // Putting back data from Ctile into C
    int counter = 0;
    for (int ip = 0; ip < min(182, 512-it);
                                    ip++)
      for (int jp = 0; jp < min(74, 512-jt);
                                    jp++)
        C[it+ip][jt+jp] =
                  *(&Ctile[0][0] + (counter++));

  } // end of j tile loop jt
} // end of i tile loop it
}
```

*Listing 3: Programul OPINCAA simplificat generat de compilatorul Connex-S LLVM din Listing-ul 2*

Obţinem de la modulul SRA invocat în modulul nostru LLVM *LoopVectorizeOpincaa*, odată ce avem o idee care sunt operaţiile vectoriale şi de coordonare pentru programul OPINCAA, că memoria folosită de buclele imbricate în această funcţie sunt 1,048,576 bytes. Aceasta într-adevăr este mărimea celor două matrice de input A şi B. Motivul pentru care nu includem de asemenea aici rezultatul matricei C este că Connex-S foloseşte pentru eficienţă unitatea funcţională hardware sum-reduction care să execute bucla cea mai internă vectorizată care calculează *dot-product-ul (produsul scalar)*, şi rezultatul reducerii este trimis direct la CPU fără a fi stocat în memoria LS.

Algoritmul nostru simplu de căutare brute-force întoarce un vector de mărimi tile de performanţă optimă (182, 74, 512). În acest vector, asociem mărimile de la stânga la dreapta cu o mărime tile 182 pentru bucla exterioară *i* până la un tile de 512 elemente pentru bucla internă k. În acest caz, ultima mărime implică faptul că noi nu facem nici un tiling pentru bucla k, care rămâne la fel ca originalul.

PPCG generează array-urile auxiliare *Atile* şi *Btile* pentru ca să începem complet în 256 KB date din array-urile originale A, respectiv B, înaintea codului actual de multiplicare.

Atunci, compilatorul LLVM Connex-S generând după copierea datelor de intrare în tile-uri şi înainte de execuţia kernel-ului apelul blocant *writeDataToConnexPartial()* transferurile I/O de la aceste array-uri tile către Connex-S cu un volum de date calculat cu modulul SRA - a se vedea Listing-ul 3. Similar, array-ul *Ctile* este copiat la sfârşitul buclei *jt* în locaţiile corespunzătoare matricei rezultat C, care este orchestrat cu faptul că LLVM adaugă mai târziu un apel *readCorrectReductionResults()* imediat înainte acestei copieri a tile-ului. Interesant, PPCG plasează copiile de la matricea A către array-ul *Atile* în afara buclei for *jt*, care reuşeşte să atingă refolosirea optimă a datelor în sensul că acelaşi bloc de date *Atile* este refolosit pentru toate valorile posibile ale lui *Btile*.

Explicăm acum câteva notaţii în kernel-ul OPINCAA în Listing-ul 3: R(0) se referă la registrul vectorial 0 al lui Connex-S, iar LS[R(3)] denotă pentru fiecare unitate de execuţie valoarea stocată la adresa din memoria LS indicată din respectivul element din R(3). Cele două bucle for C++ în kernel-ul de asamblare vectorial, *ip* şi *kStripmine*, permit lui OPINCAA să facă la runtime unroll total al codului de asamblare conţinut în corpul lor. Numele kernel-ului OPINCAA Connex-S are prefixul *allowRedefine* deoarece noi redefinim şi asamblăm kernelul pe CPU de 21 de ori în timpul execuţiei de program. Kernel-ul începe să fie executat pe accelerator numai când CPU-ul intră în metoda *executeKernel()*.

Atunci, apelul blocant al metodei *readCorrectReductionResults()* aşteaptă pentru $min(182, 512 - it) \cdot min(74, 512 - jt)$ rezultate din unitatea de reducere a lui Connex-S, pe care le folosim să actualizăm elementele corespunzătoare ale matricei *Ctile*. Acest număr de rezultate de reducere este calculat în acest exemplu, ca şi în alte cazuri, prin multiplicarea numărul de iteraţii al celor două bucle exterioare de element, dar planificăm să folosim în viitorul apropiat modulul LLVM Loopus să suporte cazuri mai

generale. Trebuie să observăm că noi nu stocăm *Ctile* în memoria LS, deoarece calculăm fiecare element al array-ului cu unitatea hardware de sum-reduction al lui Connex-S şi este trimis direct la CPU în ordinea corectă.

Dorim să subliniem că kernel-ul de asamblare este vector-length agnostic deoarece efectuăm asamblare JIT şi numărul de iteraţii ale buclei for *kStripmine* are ca parametru pe CVL, care face unroll total al codului dinăuntrul corpului. Putem vedea că codul vectorial al lui Connex-S OPINCAA foloseşte expresia C/C++ *(ip∗512)/CVL* ca un operand simbolic scalar imediat atribuit la R(3), pe care asamblorul îl translatează la runtime la o valoare concretă.

Putem observa că programul C cu tiling din Listing-ul 2 are nişte probleme să compileze cu LLVM datorită operatorilor *min*, fiind prezentat aşa pentru lizibilitate. Motivul este că şi algoritmul SRA şi kernel caching-ul lui OPINCAA nu suportă *min* ca limită a buclei. Mai exact: (i) putem consulta această limită a algoritmului SRA în [Nazaré et al., 2014]; şi (ii) dacă kernel-ul OPINCAA conţine *min* atunci caching-ul datelor binare ale kernel-ului la prima execuţie este incorect deoarece valoarea întoarsă de operatorul *min* se poate schimba într-o execuţie următoare, rezultând în diferite instrucţiuni de kernel decât cele cache-uite. Pentru a adresa acest lucru realizăm cu PPCG un fel de transformare de tip *loop unswitching* a codului din Listing-ul 2, în care noi duplicăm bucla de elemente *i* (*ip*), fiecare copie primind un operand al operatorului *min* ca limită superioară de buclă şi fiind executată condiţionat pe baza valorii predicatului *182 < 512 - it*. Când folosim compilatorul Connex-S LLVM fiecare nouă buclă *ip* devine parte a unui noi kernel OPINCAA de tiling cu propriile lui apeluri de I/O. Similar, aplicăm această transformare pentru bucla *jp* cu REPEAT. Nu prezentăm aceste versiuni corectate ale programului C cu tiling şi a codului OPINCAA C++ datorită lipsei de spaţiu, ceea ce înseamnă şi că Listing-ul 3 nu poate realiza caching-ul kernel-ului.

De asemenea recomandăm consultarea exemplului simplu de compilare fără tiling ale programului SumReduce din [Şuşu, 2019].

## 4. Experimente

Acum noi prezentăm nişte benchmark-uri corespunzătoare folosite spre exemplu în aplicaţii îmbarcate *high performance* şi de *computer vision* pe care compilatorul C optimizant mânuieşte şi evaluează performanţa codului generat de acceleratorul Connex-S.

Pentru experimente, noi folosim o platformă de dezvoltare Zedboard cu un *SoC (System On Chip)* Xilinx Zynq-7020, cu o distribuţie ArchLinux 1.4 cu kernel Linux 3.14.0 şi GCC 8.3.0 pentru ARM. Compilatorul Connex-S extinde LLVM 8.0 din martie 2019 împreună cu un modul *LoopVectorize* din LLVM 3.8 din iulie 2016, şi PPCG, cu versiunea 0.08.2. Biblioteca OPINCAA este disponibilă pentru a fi descărcată la adresa din referinţa [DCAE, 2019].

Prezentăm în Figura 2 creşterea performanţei a câtorva benchmark-uri scrise în C când rulăm pe acceleratorul Connex-S cu 128 unităţi de execuţie şi o memorie locală SPM de 256 KB pentru datele de program, sintetizat pe un Xilinx Zynq-7020 FPGA, la 100 MHz relativ la procesorul dual-core ARM Cortex A9 integrat în Zynq SoC, la 667 MHz, echipat cu un pipeline superscalar cu 8 stagii cu suport 128-bit Neon SIMD. Pentru Connex-S folosim compilatorul OPINCAA LLVM, iar pentru Cortex A9 folosim ARM GCC 8.3.0, şi pentru ambele compilatoare specificăm un nivel maxim de optimizare.

Toate benchmark-urile folosesc array-uri cu elemente de tip nativ *i16 (16-bit integer)*, dar de asemenea pentru tipurile emulate *i32 (32-bit integer)* şi *f16 (16-bit floating point)*, unde este posibil. Benchmark-urile realizează: *dot-product (DotProd)* pe array-uri cu un număr de 64K *i16* sau *f16* elemente, sau 32K *i32* elemente; sum-reducerea numărului de populaţii din cuvinte (*CtPop-Reduce*) pe un array de 128K elemente de tip *i16* - noi nu rulăm de asemenea pe *i32*, deoarece este un tip emulat, mai puţin eficient pe Connex-S; multiplicarea de matrice (*MatMulBT-128/170/256/512/1024*) pentru operanzi de mărime 128×128, 170×170, 256×256, 512×512 şi 1024×1024, a doua matrice fiind deja transpusă să permită vectorizarea; *MatMul-128*, care face de asemenea transpoziţia matricei implementată manual în limbaj de asamblare vectorială în loc de limbaj C pentru cel de-al doilea operand de intrare înaintea calculului descris anterior pentru *MatMulBT-128* pentru a urma standard BLAS; *Sum of Squared Differences (SSD)* şi *Sum of Absolute Differences (SAD)*, funcţii standard folosite spre exemplu în *computer vision* pentru detectarea trăsăturilor SIFT sau pentru estimarea mişcării [Bocchino & Adve, 2006], calculează statistici pentru toate perechile de două grupuri de 2048 decolecţii de 128 elemente de tip *i16, i32,* sau *f16*; *covariance-128* şi *correlation-128* sunt două benchmark-uri din suita PolyBench [Pouchet, 2014] care calculează covarianţa şi

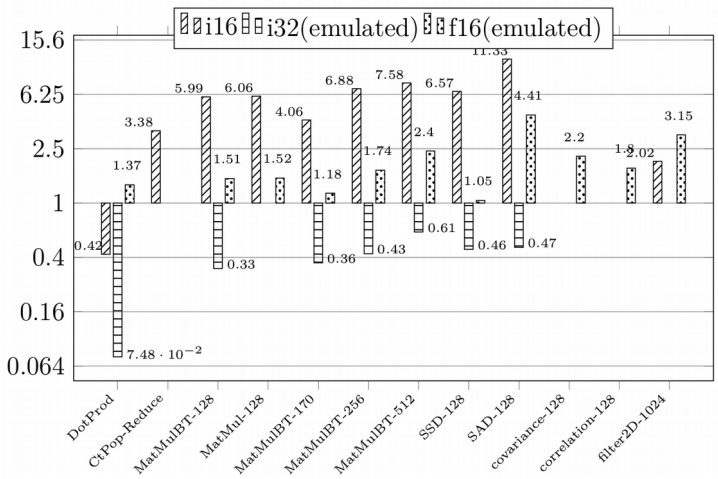


*Figura 2: Speedup-urile reprezentate semi-logaritmic pentru benchmark-urile rulate pe Connex-S cu 128 de lane-uri, la 100 de MHz relativ la un ARM Cortex A9 dual-core la 667 MHz cu un total de două unităţi Neon SIMD de 128 biţi.*

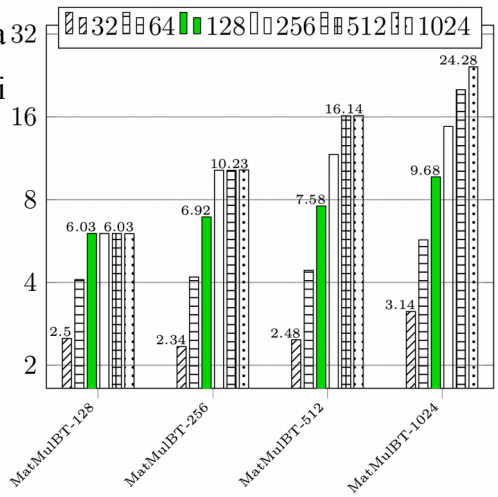


*Figura 3: Speedup-urile reprezentate semi-logaritmic pentru benchmark-urile de tip i16 rulate pe Connex-S cu un număr de lane-uri între 32 şi 1024, cu frecvenţa de 100 MHz relativ la un ARM Cortex A9 dual-core la 667 MHz cu un total de două unităţi Neon SIMD de 128 de biţi.*

corelarea a 128 de puncte de data, fiecare cu 128 de atribute de tip *f16*; filter2D-1024 realizează o transformare de CV de tip convoluție pe o imagine de 1024×768 pixeli, cu un kernel de corelare de mărime de 15×15, fiind folosit pentru a elimina zgomotul din imagini sau să detecteze trăsături sau contururi în imagini.

Kernel-urile *DotProd, CtPop-Reduce, MatMulBT-256.i16/f16* au date de intrare de exact 256 KB, mărimea memoriei SPM, dar *MatMulBT-128/170, MatMul-128, covariance-128, şi correlation-128* au mai puțin. Cu toate acestea, kernel-urile *MatMulBT-256.i32, MatMulBT-512/1024, SSD-128, SAD-128,* şi *filter2D-1024* au un footprint (consum) de memorie mai mare decât SPM-ul, deci facem tiling optim pentru ele. Nu avem optimizări de tiling a buclelor pentru programele care rulează direct pe ARMv7, deoarece ARM GCC 8.3.0 oferă puțin suport pentru ele. În plus, hardware prefetcher-ul speculativ al lui ARM este slab, nefavorizând execuția de cod cu tiling.

Benchmark-urile *MatMulBT-128/170/256.i16/i32/f16* sunt de fapt implementate cu aceeaşi funcție C sursă care acceptă două input-uri şi o matrice de output a mărimilor N×N, reprezentat ca un array 2D de lungime variabilă, disponibil în C99 şi mai noile standarde ale limbajului C. Cu toate acestea, în cazul în care trebuie să realizăm tiling pentru matricele de mărime 512×512 sau 1024×1024, momentan nu putem folosi acest kernel de mărime parametrică, în principiu deoarece numărul de iterații ale buclelor nu sunt cunoscute la compilare aşa că nu putem calcula un tiling optimal static. De asemenea, kernel-urile *DotProd* şi *CtPop-Reduce* iau la intrare date de mărime parametric, dar putem oferi la intrare pointeri în loc de array-uri 1D de lungime variabilă şi nu e nevoie de tiling deoarece datele au mărime mică.

Pentru benchmark-urile *i16*, observăm că nu putem accelera kernel-ul *DotProd*, deoarece are o intensitate aritmetică mică, şi transferurile de memorie la Connex-S devin predominante. Cu toate acestea, celelalte kernel-uri *i16* ating accelerare bună, de asemenea în parte deoarece GCC 8.3.0 nu vectorizează pentru ARM Neon testele *i16* ale noastre, ceea ce se întâmplă deoarece este considerat neprofitabilă vectorizarea sum-reducerile şi multiplicări de vectori. Observăm că benchmark-ul *MatMulBT-170* pentru tipuri *i16* şi *f16* are un speedup mai mic decât benchmark-ul similar *MatMulBT-128/256*, deoarece noi *pad*-uim cu 86 de elemente zero fiecare linie a fiecărei matrice de intrare să le facă pe amândouă de mărime 170×256 pentru a evita realizarea de reduceri parțiale. De asemenea, notăm că speedup kernel-ului *SAD-128* este mai mare decât cel al lui *SSD-128*, deoarece ARM are un predictor de salturi care eşuează des când calculează valoarea absolută, deoarece datele de intrare sunt aleatoare.

Multe dintre benchmark-urile de tip *i32* ating un speedup subunitar datorită complexității mari a operațiilor *i32* emulate pe Connex-S şi deoarece GCC vectorizează programe cu tipul *i32* pentru ARM. Ca să fim în stare să accelerăm aceste benchmark-uri *i32* pe Connex-S, acesta ar trebui să fie mai lat:

spre exemplu, *MatMulBT-512.i32* pe un Connex-S cu 512 lane-uri ar trebui să atingă un speedup, care este de aproape trei ori mai bun decât pentru 128 de lane-uri.

Nu putem neglija de asemenea speedup-ul arhitectural, care este raportul de îmbunătățire a numărului de ciclii executați, în cazul nostru este mai mare decât speedup-urile reportate în Figurile 2 şi 3 practic de 6.67 ori, dat fiind că frecvenţa CPU-ului şi a procesorului Connex-S sunt 667 şi 100 MHz.

Experimentăm o accelerare decentă de benchmark-uri *f16* deoarece ARMv7 nu suportă nici el tipul *f16* nativ, deci trebuie să îl convertească către *f32* pentru a executa operații native şi după aceea să revină la *f16*, şi aceste operații de conversie au un cost mare. Un motiv mai puțin important este faptul că GCC 8.3.0 nu poate vectoriza operații floating-point pentru ARM Neon.

Pentru aceste benchmark-uri, scrierea consecutivă a două blocuri de 128 KB din RAM-ul de sistem către memoria LS ia 1.265 ms. De asemenea, asamblarea unui kernel OPINCAA ia ca timp între 0,01-134,6 ms, kernel-ul *MatMulBT-256.f16* atingând maximul de 308 Kinstrucțiuni. Pentru a amortiza acest cost noi cache-uim datele binare de instrucțiuni pe care le trimitem către Connex-S, pentru aceleaşi mărimi de date de intrare, când rulăm kernel-ul de mai multe ori – de 100 sau de 1000 de ori în experimentele noastre. De asemenea putem precomputa datele binare ale kernel-ului înainte de rularea efectivă, dar o astfel de procedură este complicată şi normal nu aduce nici un avantaj serios dacă este să comparăm cu caching-ul atunci când rulăm un kernel de mai multe ori.

Codul de asamblare vectorial generat este optim pentru aproape toate benchmark-urile, mai puțin pentru *MatMulBT-128.i16, SSD-128.i16,* şi *SAD-128.i16*, pentru care putem atinge un speedup de 1.072, 1.072, respectiv 1.049 ori mai mare. Pentru aceasta, noi trebuie să evităm a executa o nenecesară operație de acumulare de adunare vectorială (a se vedea Listing-ul 3) pentru calcularea dot-product-ului, deoarece numărul de iterații ale buclei vectorizate, 128, este egal în experimente cu CVL, lungimea vectorială a lui Connex-S. Similar, pentru *MatMulBT-128.f16, SSD-128.f16* şi *SAD-128.f16* noi scoatem instrucțiunea vectorială **add.f16**, care ia mult timp deoarece codul de asamblare vectorial are 500-700 de instrucțiuni, ceea ce creşte performanța de 1.66x, 1.4x, respectiv 1.6x. Planificăm să adăugăm această optimizare în compilator scoțând **add**-ul vectorial de acumulare în viitor. *MatMulBT-170/256* nu poate beneficia de această optimizare deoarece numărul de iterații ale buclei interne este diferită de CVL. Benchmark-urile *MatMulBT-128.i32, SSD-128.i32* şi *SAD-128.i32* nu pot fi optimizate deoarece acumularea este efectuată de fapt pe vectori de 64 de elemente de tip *i32* deci fiecare dot-product efectuează două acumulări.

Putem rula acelaşi program generat C++ OPINCAA pe sisteme cu acceleratoare Connex-S cu diferite lungimi vectoriale deoarece numărul de lanes CVL este o variabilă OPINCAA de program de mediu şi noi efectuăm asamblare vectorială JIT. În Figura 3 prezentăm speedup-urile obținute prin generarea de programe OPINCAA când rulăm pe procesorul Connex-S de diferite lungime vectoriale. De fapt,

deoarece timpul de execuţie pe CPU nu variază cu CVL, aceste grafuri arată variaţia timpului de execuţie pe Connex-S când CVL variază. Menţionăm că experimentele cu 256, 512, şi 1024 de lane-uri sunt efectuate cu simulatorul Connex-S OPINCAA deoarece FPGA-ul Xilinx Zynq-7020 nu poate acomoda aceste largi variante de procesoare Connex-S. Noi vedem cum performanţa acestor benchmark-uri variază când creştem lungimea vectorială a acceleratorului. Acest trend nu este linear dar asimptotic, vizibil în special pentru *MatMulBT-1024*, datorită costului de comunicare I/O şi a imposibilităţii de a accelera prologul şi epilogul buclei vectorizate când creştem CVL. Observăm că pentru *MatMulBT-128/256/512* accelerarea stagnează după 128, 256, respectiv 512 lane-uri, deoarece numărul de iteraţii ale respectivei bucle vectorizate devine mai mică decât lungimea vectorială, şi restul de lane-uri rămân nefolosite.

# 5. Generare de cod optimizat folosind matrice rare pentru transformări de computer vision

La sfârşitul tezei am inclus un capitol legat de aplicaţii de computer vision cu potenţial de a fi accelerate pe procesorul Connex-S.

Transformări de imagine moderne de *computer vision* cu stagii complexe pot beneficia de date de intrare rare. Cu toate acestea, aceste rutine sunt dificile de implementat eficient cu matrice rare.

De aceea, noi propunem un compilator cu un *limbaj de domeniu specific* la intrare (în englezeşte, *Domain-Specific Language, DSL*) pe care specialiştii de computer vision pot să îl folosească. Acest program citeşte o funcţie matematică descriind o transformare, creând o relaţie între pixelii imaginii de input şi output, împreună cu un template generic de cod C, şi generează din ele o implementare C eficientă folosind matrice rare. Compilatorul realizează o nouă optimizare de cod care să crească performanţa astfel încât să traversăm în-ordine imaginea de input rară. Pentru aceasta compilatorul DSL are nevoie să efectueze o optimizare simbolică matematică pe ecuaţiile funcţiei specificate de transformarea în-ordine pentru a inversa funcţia.

Folosim compilatorul nostru DSL pentru a ne ajuta să dezvoltăm aplicaţia de computer vision cu matrice rare efectuând alinieri de imagini. Reuşim să obţinem o creştere medie a performanţei secvenţiale de 8.32 ori pe un CPU x86 la 2.66 GHz când folosim matrice rare în diverse rutine folositoare de *computer vision* decât atunci când folosim imagini dense.

De asemenea, accelerăm nişte computaţii pe procesorul larg vectorial Connex-S realizând o creştere minoră de performanţă relativ doar la un singur CPU. De asemenea, ne uităm să îmbunătăţim performanţa folosind o simplă reprezentare de matrice rare cu blocuri începând de la observaţia că datele rare în cadru sunt reprezentate, cu discuri cu raza de 2.5 (sau 3) pixeli.

Scopul final al acestui capitol pe lângă faptul de a documenta o metodologie matematică interesantă şi simplă de execuţie eficientă a calculelor matriceale de tip *stencil* sau *map* [McCool_2012] pentru *computer vision* pornind de la specificaţii DSL este să arătăm de asemenea că astfel de computaţii pot fi bine accelerate pe procesoare Connex-S. Includem aceste rutine de calcul în biblioteca de runtime a lui Connex-S, lăsând compilarea din C către Connex-S ca muncă pe viitor.

# Bibliografie

2020. The Polyhedral Model, http://polyhedral.info.

Bîră, Călin, Petrică, Lucian, & Hobincu, Radu. 2013. OPINCAA: A Lightweight and Flexible Programming Environment For Parallel SIMD Accelerators. Romanian Journal of Information Science and Technology, 16(4).

Bocchino, Jr., Robert L., & Adve, Vikram S. 2006. Vector LLVA: A Virtual Vector Instruction Set for Media Processing. Pages 46–56 of: Proceedings of the 2nd International Conference on Virtual Execution Environments. VEE '06. New York, NY, USA: ACM.

Chandrakasan, A. P., Sheng, S., & Brodersen, R. W. 1992. Low-power CMOS Digital Design. IEEE Journal of Solid-State Circuits, 27(4), 473–484.

Şuşu, Alexandru E. 2020. A Vector-Length Agnostic Compiler for the Connex-S Accelerator with Scratchpad Memory. ACM Trans. Embed. Comput. Syst., 19(6).

Şuşu, Alexandru Emilian. 2019. Compiling Efficiently with Arithmetic Emulation for the Custom-Width Connex Vector Processor. Pages 1:1–1:8 of: Proceedings of the Workshop on Programming Models for SIMD/Vector Processing. WPMVP'19. New York, NY, USA: ACM.

DCAE. 2019. The Connex-S OPINCAA library, cod sursă disponibil la http://gitlab.dcae.pub.ro/research/opincaa .

Kennedy, Ken, & Allen, John R. 2002. Optimizing Compilers for Modern Architectures: A Dependence-based Approach. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Kodukula, Induprakas, Ahmed, Nawaaz, & Pingali, Keshav. 1997. Data-centric Multi-level Blocking. Pages 346–357 of: Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation. PLDI '97. New York, NY, USA: ACM.

Kozyrakis, Christoforos. 2002. Scalable Vector Media-Processors for Embedded Systems. Ph.D. thesis, University of California, Berkeley. AAI3063439.

Kozyrakis, Christoforos E., & Patterson, David A. 2003. Scalable Vector Processors for Embedded Systems. IEEE Micro, 23(6), 36–45.

Lattner, Chris, & Adve, Vikram. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. Pages 75– of: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization. CGO '04. Washington, DC, USA: IEEE Computer Society.

Lin, Haibo, Liu, Tao, Li, Huoding, Chen, Tong, Renganarayana, Lakshminarayanan, O'Brien, John Kevin, & Shao, Ling. 2010. DMATiler: Revisiting Loop Tiling for Direct Memory Access. Pages 559–560 of: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques. PACT '10. New York, NY, USA: ACM.

McCanne, Steven, & Jacobson, Van. 1993. The BSD Packet Filter: A New Archi- tecture for User-level Packet Capture. Pages 2–2 of: Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings. USENIX'93. Berkeley, CA, USA: USENIX Association.

McCool Michael, Reinders James, Robison Arch. Structured Parallel Programming: Patterns for Efficient Computation, 2012, Morgan Kaufmann Publishers Inc.

Mendonça, Gleison, Guimarães, Breno, Alves, Péricles, Pereira, Márcio, Araújo, Guido, & Pereira, Fernando Magno Quintão. 2017. DawnCC: Automatic Annotation for Data Parallelism and Offloading. ACM Trans. Archit. Code Optim., 14(2), 13:1–13:25.

Naishlos, Dorit. 2004. Autovectorization in GCC. Proceedings of the 2004 GCC Developers Summit.

Nazaré, Henrique, Maffra, Izabela, Santos, Willer, Barbosa, Leonardo, Gonnord, Laure, & Quintão Pereira, Fernando Magno. 2014. Validation of Memory Accesses Through Symbolic Analyses. Pages 791–809 of: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications. OOPSLA '14. New York, NY, USA: ACM.

Nuzman, Dorit, & Henderson, Richard. 2006. Multi-platform Auto-vectorization. Pages 281–294 of: Proceedings of the International Symposium on Code Generation and Optimization. CGO ’06. Washington, DC, USA: IEEE Computer Society.

Patterson, David A., & Hennessy, John L. 2017. Computer Organization and Design RISC-V Edition: The Hardware Software Interface. 1st edn. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. Pouchet, L. N. 2014. PolyBench: The Polyhedral Benchmark Suite.

Ştefan, Gheorghe M. 2015. The Connex-S Instruction Set Architecture. The Linux Kernel Archives. 2014. https://www.kernel.org/doc/Documentation/networking/filter.txt.

Waeijen, Luc, She, Dongrui, Corporaal, Henk, & He, Yifan. 2015. A Low-Energy Wide SIMD Architecture with Explicit Datapath. J. Signal Process. Syst., 80(1), 65–86.