UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Optimizarea algoritmilor computationali, folosind instructiuni SIMD

Structura sistemelor de calcul

Autor: Saca Victor-Valentin

Grupa: 30238

Indrumator:

Facultatea de Automatica si Calculatoare

15 Ianuarie 2025

Cuprins

1 Introducere

Anumiti algoritmi computationali au dezavantajul de a fi ineficienti, si pot scade performanta unui program, sau unui sistem. O tehnica de optimizare a acestora este gasirea unei variante cu o complexitate in timp mult mai mica (liniar, sau logaritmic (sau nlog(n)). Dar, nu tot timpul se pot gasi aceste solutii. Mai ales cand vorbi despre sortari de date, unde complexitatea nu poate scadea sub n*log(n) (n - numarul de elemente).

Astfel, o solutie care poate creste performanta semnificativ este optimizarea algoritmilor folosind instructiuni single instruction stream, multiple data stream.

Veti putea observa la partea de Rezultate experimentale, cum pentru un algoritm considerat foarte ineficient, timpul de executie al acestuia se injumatateste pentru oricare ar fi numarul de inputuri.

2 Fundamentare teoretica

2.1 Stooge sort

Principiul Stooge sort consta in compararea elementelor din capete si permutarea acestora pe baza comparatiei, iar dupa urmeaza apelurile recursive pe primele 2 treimi a listei, ultimele 2 treimi si din nou primele 2 treimi.[1]

2.2 ARM NEON

ARM NEON este o extensie hardware pentru procesoarele ARM folosita de cele mai multe ori in aplicatii multimedia, procesarea semnalelor si orice implica operatii pe vectori de date.[2]

Aceasta extensie s-a folosit cu ajutorul librariei C $< arm\_neon.h >$

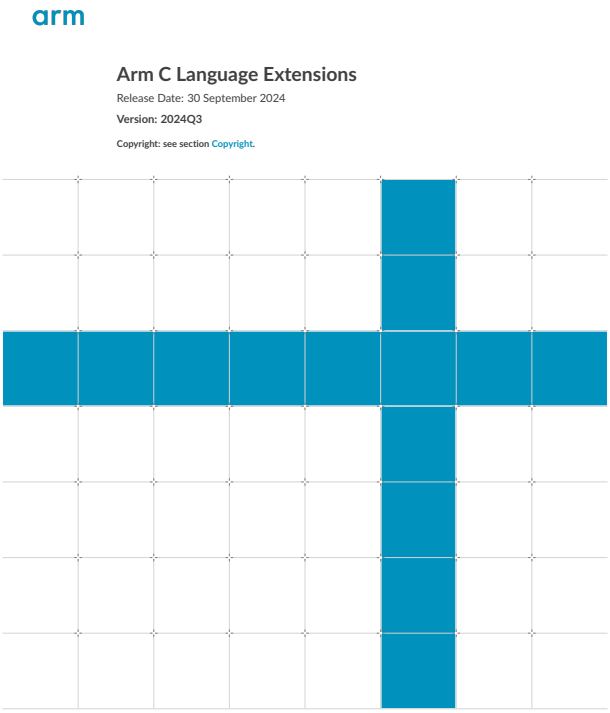
2.3 Optimimizarea cu ajutorul instructiunilor SIMD

Introducerea vectorizarii intr-un algoritm ca Stooge sort reduce timpul de executie pentru date mari, deoarece operatiile de comparare si interschimbare se realizeaza simultan pe blocuri de 4 elemente. Mai mult, se reduce incarcarea procesorului prin folosirea registrelor vectorilor.

O alta solutie posibila ar fi fost autovectorizarea. Aceasta este tot vectorizare, dar facuta de compilator in timpul executarii codului. Aceasta se activeaza doar specificand anumite flag-uri in momentul compilarii

Pentru realizarea proiectului, documentarea s-a facut din manualul oficial ARM C language extensions[3], dar mai ales din manualul setului de instructiuni: `https://arm-software.github.io/acle/neon_intrinsics/advsimd.html#list-of-intrinsics`

[1] geeks for geeks : Stooge Sort - `https://www.geeksforgeeks.org/stooge-sort/`
[2] arm.com - technologies - `https://www.arm.com/technologies/neon`
[3] Arm C Language Extensions -



3 Proiectare si implementare

3.1 Metoda experimentala utilizata

Proiectul este implementat software, folosind limbajul C cu extensiile ARM NEON pentru optimizarea SIMD.

Pentru testare s-a folosit placa FPGA Zybo-20 cu procesor ARM, cu ajutorul mediului de dezvoltare Vitis : Am creeat un IP pentru Zynq, am creeat HDL wrapper, iar apoi am generat bitstream pentru mediul de dezvoltare vitis, unde am incarcat algoritmul.

3.2 Solutia aleasa

Pentru acest proiect, am ales sa folosesc vectorizare partiala pentru blocuri mai mari de 4 elemente, deoarece pentru blocuri mici de date, nu va exista o diferenta de performanta, si am ales sa optimizez cazuri de dimensiuni mari al listei (vector). Astfel, vectorul se imparte in blocuri de 4 elemente, pentru ca mai apoi sa se aplice interschimbari acolo unde este cazul, simultan pe acele blocuri.

3.3 Schema bloc

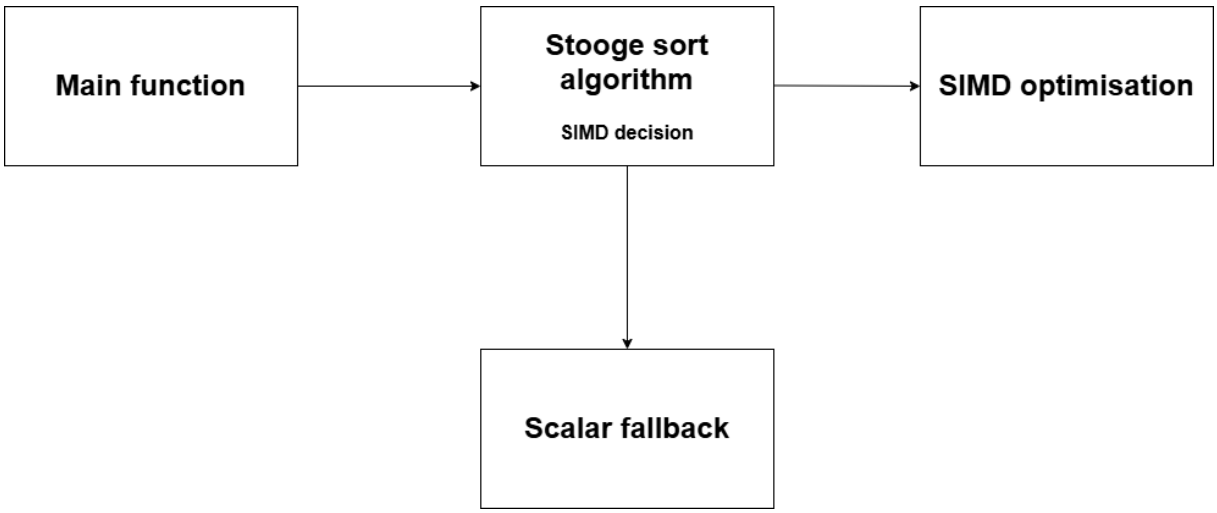


Figura 1:
- Main function : genereaza vectorul, si afiseaza rezultatul impreuna cu timpul de executie
- Stooge sort algorithm SIMD decision : implementeaza algoritmul de baza si decide daca utilizeaza instructiuni SIMD sau cade in scalar fallbak
- SIMD optimisation : Optimizeaza segmente mari ale vectorului cu instructiunile NEON.
- Scalar fallback : Proceseaza segmentele mici ale array-ului

3.4 Implementarea fiecarui modul:

3.4.1 Main Function:

```
int main() {
    int r = 1000;
    int32_t a[r];
    for (int i = r; i > 0; i--)
        a[r - i] = i;

    int l = 0;

    printf("The array before sorting\n");
    for (int i = 0; i < r; i++)
        printf("%ld ", a[i]);

    clock_t exec_time = clock();
    stoogeSort(a, l, r - 1);
    exec_time = clock() - exec_time;
    double time_taken = exec_time / (double) CLOCKS_PER_SEC;

    printf("\nThe array after sorting\n");
    for (int i = 0; i < r; i++) {
        printf("%ld ", a[i]);
    }
```

```
    printf("\nExecution time: %lf\n", time_taken);
    return 0;
}
```

Functia main genereaza un vector de 5000 de elemente, ordonate descrescator care urmeaza a fi sortate crescator.

De asemenea, cu functia clock() din $< time.h >$ se calculeaza timpii de executie de la pornirea algoritmului stooge sort.

3.4.2 Stooge sort algorithm, SIMD decision

```
void stoogeSort(int32_t* a, int l, int r) {
    if (l >= r)
        return;

    int n = r - l + 1;

    if (n >= 4) {
        //simd optimisation
    }

    //scalar fallback

    if (n > 2) {
        int d = n / 3;

        stoogeSort(a, l, r - d);
        stoogeSort(a, l + d, r);
        stoogeSort(a, l, r - d);
    }
```

Algoritmul de baza imparte vectorul in 3 parti in mod recursiv. La inceput se aplica pe primele 2 treimi din vector, apoi pe ultimele 2 treimi, iar la final pe primele 2.

De asemenea, in functie de numarul de elemente din vector la care s-a ajuns, se decide daca operatiile se fac simultan sau nu pe blocuri de cate 4 din vector

3.4.3 SIMD optimisation

```
        int32_t* a_l = (int32_t*) (a + l);
        int32_t* a_r = (int32_t*) (a + r - 3);

        int32x4_t left_block = vld1q_s32(a_l);
        int32x4_t right_block = vld1q_s32(a_r);

        int32x4_t min_vals = vminq_s32(left_block, right_block);
        int32x4_t max_vals = vmaxq_s32(left_block, right_block);

```

```
        vst1q_s32(a_l, min_vals);
        vst1q_s32(a_r, max_vals);
```

Aceasta etapa reprezinta optimizarea folosind SIMD efectiva.

- Se compara si se schimba valori intre primele si ultimele 4 elemente ale vectorului.

-Se continua prin apeluri recursive din Stooge sort pentru subvectori mai mici

- Daca s-a ajuns la mai putin de 4 elemente se continua cu Fallback scalar.

3.4.4 Scalar Fallback

```
if (n < 4) {
        for (int i = l; i <= r; i++) {
            for (int j = l; j <= r - (i - l); j++) {
                if (a[j] > a[j + 1]) {
                    int temp = a[j];
                    a[j] = a[j + 1];
                    a[j + 1] = temp;
                }
            }
        }
        return;
    }
```

La aceasta etapa se vor procesa partile din array cu numar de elemente mai mici de 4. Se itereaza in vectorul ramas, si se interschimba acolo unde este cazul pentru a "grabi" procesul de sortare. La fel cum am mai mentionat, la aceasta etapa nu se utilizeaza optimizari SIMD.

3.4.5 Detalii de implementare

Algoritmul primeste ca intrare un array si limitele acestuia si vectorul se va transforma in memorie, unde va fi stocat vectorul sortat.

-Arhitectura software:

Comunicare intre module se face doar prin functii, neexistand o interfata pentru utilizator, proiectul fiind demonstrat prin afisarea consolei.

-Biblioteci utilizate:

$< arm\_neon >$ - pentru instructiunile SIMD.
$< stdint.h >$ - pentru tipurile de date portabile[1].
$< stdio.h >< time.h >$ - pentru masurarea timpului de executie si afisarea rezultatului.

[1] - include tipuri de date de tipul int32_t care specifica exact lungimea unui intreg.

-Cerinte de sistem

- Procesor ARM cu suport pentru NEON (de exemplu Cortex A9)

- Un compilator compatibil cu NEON.

- Alternative abandonate:

Vectorizarea completa: Aceasta nu a fost implementata deoarece ar fi crescut complexitatea codului si ar fi introdus overhead suplimentar pentru cazurile cu array-uri mici.

Alte instrucțiuni SIMD: S-a renuntat la utilizarea unor instructiuni mai avansate deoarece nu au imbunatatit performanta semnificativ.

4 Rezultate experimentale

Din cauza unei lipse de placi cu procesor ARM, am generat si o versiune a programului pentru procesor intel. Instructiunile echivalente sunt:

INTEL	ARM	DESCRIPTION
__m128i	int32_t/int32x4_t	Storing 4 elements of 32 bits
_mm_loadu_si128	vld1q_s32	Load 4 elements from an array
_mm_min_epi32	vminq_s32	Compute the element-wise minimum of two vectors of 32-bit signed integers.
_mm_max_epi32	vmaxq_s32	Compute the element-wise maximumof two vectors of 32-bit signed integers.
_mm_storeu_si128	vst1q_s32	store a 128-bit vector of four elements intoco SIMD register into memory

Pentru a putea compara, si a vedea optimizarea executiei programului, am realizat si varianta originala stooge sort fara niciun fel de optimizare.

Astfel pe un procesor Intel cu frecventa de 2.50 ghz, unde programul s-a rulat de 5 ori, de fiecare data cu un numar de elemente mai mare:

Pentru 1000 de elemente:

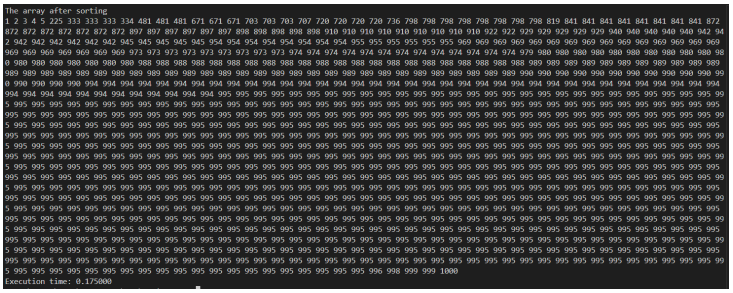


Figura 2: SIMD : 0.175

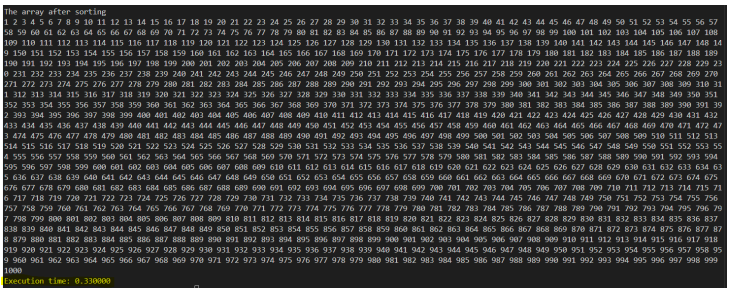


Figura 3: N/A : 0.33

2000 elemente:

Figura 4: SIMD : 1.86

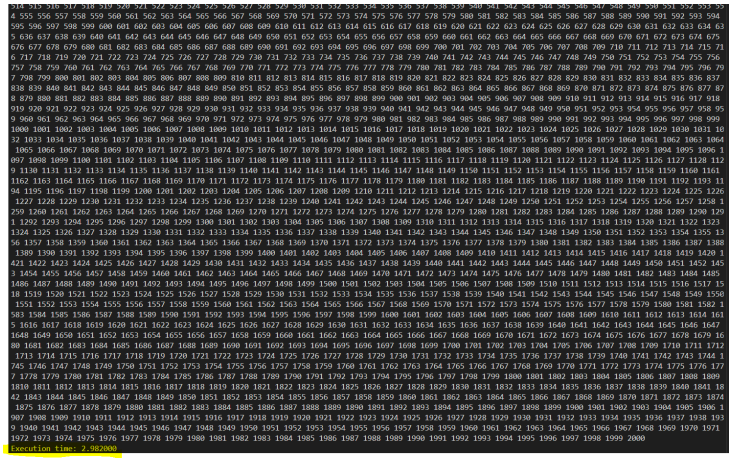


Figura 5: N/A : 2.982

3000 de elemente:

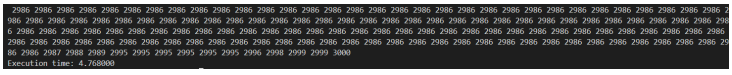


Figura 6: SIMD: 4.768

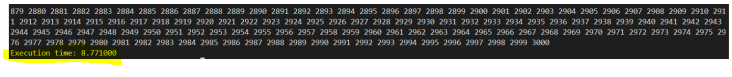


Figura 7: N/A : 8.771

4000 de elemente:

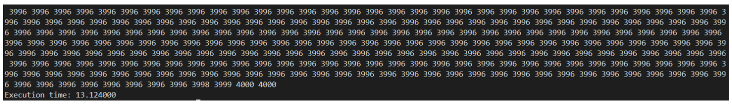


Figura 8: SIMD: 13.124

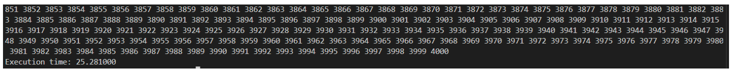


Figura 9: N/A : 25.281

5000 de elemente:

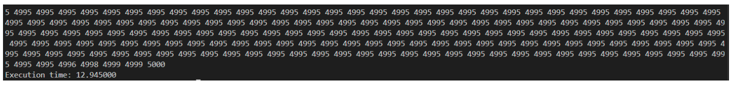


Figura 10: SIMD : 12.945

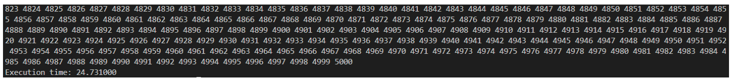


Figura 11: N/A : 24.731

Dupa cum se observa, timpul de executiei al programului care foloseste instructiuni SIMD este aproape jumatate fata de cel care nu foloseste niciun fel de optimizare.

5 Concluzii

Contribuțiile originale

1. **Integrarea flexibila a tehnicilor clasice cu cele SIMD**: fallback-ul pe secvente scalare pentru dimensiuni mici ale vectorilor a permis o implementare adaptabila.
2. **Analiza comparativa**: analiza rezultatelor obtinute a demonstrat eficiența solutiei SIMD fata de implementarea clasica.

Avantaje si dezavantaje ale proiectului

Avantaje:

- Performanța îmbunătățită pentru vectori de dimensiuni mari datorita paralelizarii.
- Flexibilitate în abordare: functioneaza atat pe date mici, cat si pe date mari.
- Utilizarea instructiunilor SIMD permite exploatarea optima a resurselor hardware disponibile.

Dezavantaje:

- Algoritmul Stooge Sort, chiar si optimizat, ramane mai puțin eficient decat metodele de sortare standard precum QuickSort sau MergeSort.
- Codul bazat pe NEON este specific arhitecturii ARM și necesita modificări pentru portare pe alte platforme.
- Performanta poate să nu fie optimă pentru dimensiuni mici ale vectorilor din cauza costului instrucțiunilor SIMD.

Aplicatii ale proiectului **Optimizarea altor algoritmi**: metodologia poate fi aplicata in optimizarea altor algoritmi ce implica sortarea sau procesarea intensiva a datelor.

Dezvoltari viitoare

1. **Generalizarea algoritmului**: adaptarea implementarii pentru arhitecturi diferite folosind alte seturi de instructiuni SIMD (e.g., AVX pentru x86).
2. **Compararea cu alte tehnici**: integrarea si compararea performantei cu algoritmi hibrizi sau paraleli.
3. **Aplicații în timp real**: testarea implementării în sisteme care necesita procesare rapida a datelor, cum ar fi aplicatiile embedded sau cele de procesare de semnal.
4. **Profilarea performantei**: utilizarea unor instrumente avansate de analiza pentru identificarea si eliminarea bottleneck-urilor din implementare.