**Kernel optimizat pentru extragerea punctelor caracteristice dintr-o amprentă**

# Descrierea proiectului

Acest proiect își propune dezvoltarea unui kernel optimizat pentru extragerea caracteristicilor (minutiae) din imagini cu amprente, utilizând algoritmi de prelucrare a imaginilor și accelerare hardware prin extensiile SIMD Neon, specifice arhitecturii ARM.

Proiectul a fost testat pe un Raspberry Pi 4B, ce permite utilizarea acestor instrucțiuni pentru optimizarea performanței.

# Pipeline de Procesare

1. **Preprocesare:**

* Citirea imaginii grayscale (cv::IMREAD\_GRAYSCALE)
* Îmbunătățirea contrastului folosind Contrast Limited Adaptive Histogram Equalization (CLAHE)
* Aplicarea unui Adaptive Threshold care păstrează detaliile fine și curbele amprentei. Rezulta o imagine alb-negru pentru prelucrare:

cv::Ptr<cv::CLAHE> clahe = cv::createCLAHE(2.0, cv::Size(8, 8));

clahe->apply(img, enhanced);

cv::adaptiveThreshold(enhanced, binary, 255, cv::ADAPTIVE\_THRESH\_GAUSSIAN\_C, cv::THRESH\_BINARY\_INV, 11, 2);

* Partea aceasta este identică pentru ambele versiuni.

1. **Skeletonizare:**

Reducerea imaginii binare la un "roadmap" al ridurilor folosind algoritmul de thinning Zhang-Suen, astfel încât forma de bază a amprentei să fie păstrată.

1. Funcția de bază (identică logic):

Condiții Zhang-Suen:

* Număr de vecini activi între 2 și 6,
* Număr de tranziții 0→1 în jurul pixelului să fie exact 1,
* Teste logice pe un subset de vecini pentru păstrarea structurii.

1. Diferența majoră: Accesul la vecini

Varianta neoptimizată (get\_neighbors):

std::vector<int> get\_neighbors(const cv::Mat& img, int i, int j) {

return {

img.at<uchar>(i - 1, j) > 0 ? 1 : 0,

img.at<uchar>(i - 1, j + 1) > 0 ? 1 : 0,

img.at<uchar>(i, j + 1) > 0 ? 1 : 0,

img.at<uchar>(i + 1, j + 1) > 0 ? 1 : 0,

img.at<uchar>(i + 1, j) > 0 ? 1 : 0,

img.at<uchar>(i + 1, j - 1) > 0 ? 1 : 0,

img.at<uchar>(i, j - 1) > 0 ? 1 : 0,

img.at<uchar>(i - 1, j - 1) > 0 ? 1 : 0

};

}

Varianta optimizată (get\_neighbors\_neon):

uint8x8\_t neighbors\_raw = vcreate\_u8(

((uint64\_t)img.at<uchar>(i - 1, j) << 0) |

((uint64\_t)img.at<uchar>(i - 1, j + 1) << 8) |

...

);

uint8x8\_t thresholded = vcgt\_u8(neighbors\_raw, vdup\_n\_u8(0));

Diferență:

* NEON încarcă și compară toți cei 8 vecini simultan folosind vectori de 8 byte (uint8x8\_t)
* Reducerea dramatică a apelurilor img.at<> => eliminare de overhead

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Neoptimizat** | **NEON** |
| **Acces vecini** | Serial, pixel cu pixel | Paralel, 8 vecini simultan |
| **Tip operații** | at<uchar>, std::vector<int> | vcreate\_u8, vcgt\_u8, thresholded[k] |
| **Iterații** | 2 sub-iterări / ciclu | Identic, dar mai rapid |
| **Praguri** | Binare manual (0/1) | Realizat automat prin vcgt\_u8 |

1. Extragerea minutiae-lor: Identificarea punctelor caracteristice (terminații și bifurcații) folosind metoda crossing number.

**Criterii:**

* Ending: o tranziție 0→1
* Bifurcație: 3 tranziții
* Sumă vecini > 0 (pentru ending)

1. Vizualizare: Marcarea punctelor detectate pe imagine cu roșu (bifurcații) și verde (terminații).

# Comparatie Tehnica: optimizare NEON vs Non-NEON

Au fost implementate doua variante ale algoritmului: una de baza (fara optimizari) si una optimizata folosind NEON. Masuratorile au fost facute pe un subset de 500 de iteratii asupra aceleiasi imagini cu o amprenta din datasetul SOCOFing.

## Timpi de executie

|  |  |  |
| --- | --- | --- |
| **Varianta** | **Timp minim (secunde)** | **Timp maxim (secunde)** |
| **NEON (optimizata)** | 6.72 s | 7.18 s |
| **Non-NEON (neoptimizata)** | 43.87 s | 44.35 s |

Diferenta clara de performanta se datoreaza folosirii instructiunilor SIMD NEON in prelucrarea pixelilor. Acestea permit procesarea paralela a mai multor pixeli intr-o singura instructiune CPU, ceea ce reduce drastic timpul total de executie.

## Implementare si Optimizari

În varianta optimizată, am folosit extensiile NEON SIMD disponibile pe procesoarele ARM (precum cel de pe Raspberry Pi 4B), pentru a accelera procesarea imaginilor binare. Aceste optimizări au fost aplicate în special asupra etapelor de skeletonizare (thinning) și extragere a caracteristicilor.

### Optimizarea procesării pixelilor

În loc să procesăm fiecare pixel individual, am folosit regiștri NEON de 128 biți pentru a procesa 16 pixeli simultan. De exemplu:

uint8x16\_t current = vld1q\_u8(&img\_row[j]);

// incarca 16 pixeli intr-un vector

uint8x16\_t neighbor\_up = vld1q\_u8(&img\_row\_up[j]);

uint8x16\_t neighbor\_down = vld1q\_u8(&img\_row\_down[j]);

uint8x16\_t vertical\_and = vandq\_u8(neighbor\_up, neighbor\_down);

// AND pe 16 pixeli

uint8x16\_t mask = vcgtq\_u8(vertical\_and, vdupq\_n\_u8(0));

// comparatie > 0

Aici se realizează în paralel verificări logice și comparări pe 16 pixeli, în loc de 16 operații separate.

### Eliminarea ramificărilor (if-uri)

În varianta neoptimizată, avem multe condiții de forma:

if (neighbors[0] \* neighbors[2] \* neighbors[4] == 0) {

// pastreaza pixelul

}

În varianta optimizată NEON, am convertit aceste condiții în operații logice directe:

uint8x16\_t cond1 = vandq\_u8(neigh0, neigh2);

uint8x16\_t cond2 = vandq\_u8(neigh4, neigh6);

uint8x16\_t combined = vandq\_u8(cond1, cond2);

uint8x16\_t result = vbicq\_u8(input, combined);

// vector bit clear de input AND (NOT combined)

### Activarea NEON în CMake

Pentru ca aceste optimizări să fie active la compilare, în fișierul CMakeLists.txt am adăugat:

set(CMAKE\_C\_FLAGS "${CMAKE\_C\_FLAGS} -mfpu=neon -O3")

Aceasta instruire spune compilatorului sa genereze cod care foloseste extensiile NEON si optimizare la nivel maxim (-O3).

## Ce s-a masurat concret?

Pentru ambele variante s-a masurat timpul necesar rulării complete a algoritmilor de extragere a caracteristicilor (mai concret algoritmii de skeletonizare si extragere minutiae) de 500 de ori consecutiv pe aceeasi imagine de input.

Imaginea folosita ca input a fost:

*/home/user/PDS/neon/input\_images/100\_\_M\_Left\_index\_finger.BMP*

La final, se produce o imagine cu amprenta subtiata, pe care sunt desenate cu verde terminatiile liniilor amprentei si cu rosu bifurcatiile acesteia.

# Măsurători de performanță

Pentru a evalua eficiența implementării algoritmului de subțiere Zhang-Suen împreună cu extragerea de minutii, s-a realizat o măsurare precisă a timpului de execuție folosind biblioteca standard C++ (<chrono>), cunoscută pentru precizia sa ridicată în analiza performanței aplicațiilor.

Mai jos este prezentată secvența de cod utilizată pentru măsurare:

using clock = std::chrono::high\_resolution\_clock;

cv::Mat last\_thinned;

std::vector<Minutia> last\_minutiae;

auto start = clock::now();

for (int k = 0; k < 500; ++k) {

last\_thinned = zhang\_suen\_thinning(binary\_image, 20);

last\_minutiae = extract\_minutiae(last\_thinned);

}

auto end = clock::now();

std::chrono::duration<double> elapsed\_seconds = end - start;

* **std::chrono::high\_resolution\_clock** este folosit pentru a obține cel mai precis ceas disponibil pentru sistem, cu granularitate de ordinul nanosecundelor.
* start și end marchează punctele de început și sfârșit ale execuției blocului de cod ce include:
  + Aplicarea subțierii Zhang-Suen (zhang\_suen\_thinning)
  + Extragerea minutiei (extract\_minutiae)
* Aceste două funcții sunt apelate de 500 de ori, pentru a asigura o medie temporală stabilă, reducând influențele variațiilor sistemului (precum alocări de memorie, sarcini OS).
* elapsed\_seconds reprezintă timpul total de execuție pentru cele 500 de runde, exprimat în secunde, folosind **std::chrono::duration<double>**.

**De ce se folosește această metodă:**

Măsurarea pe un singur cadru poate fi instabilă din cauza fluctuațiilor resurselor hardware.

Executarea repetitivă (500 de runde) oferă o estimare mai precisă a performanței algoritmului, permițând comparații relevante între versiuni optimizate (NEON) și neoptimizate (plain C++).

Acest model poate fi extins pentru a calcula și timpul mediu per execuție, astfel:

double average\_time = elapsed\_seconds.count() / 500.0;

# Rezultatele rularii

Imaginea finala salvata dupa proces contine:

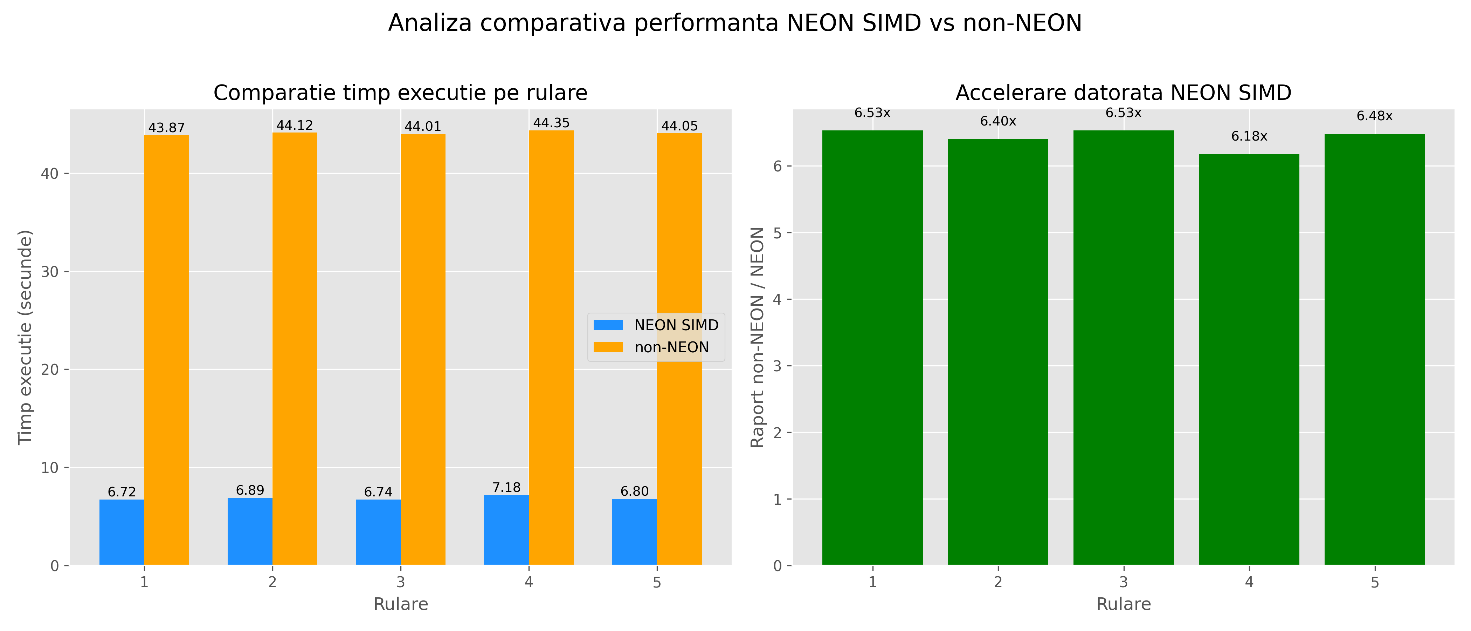
- Amprenta subtiata (skeleton)

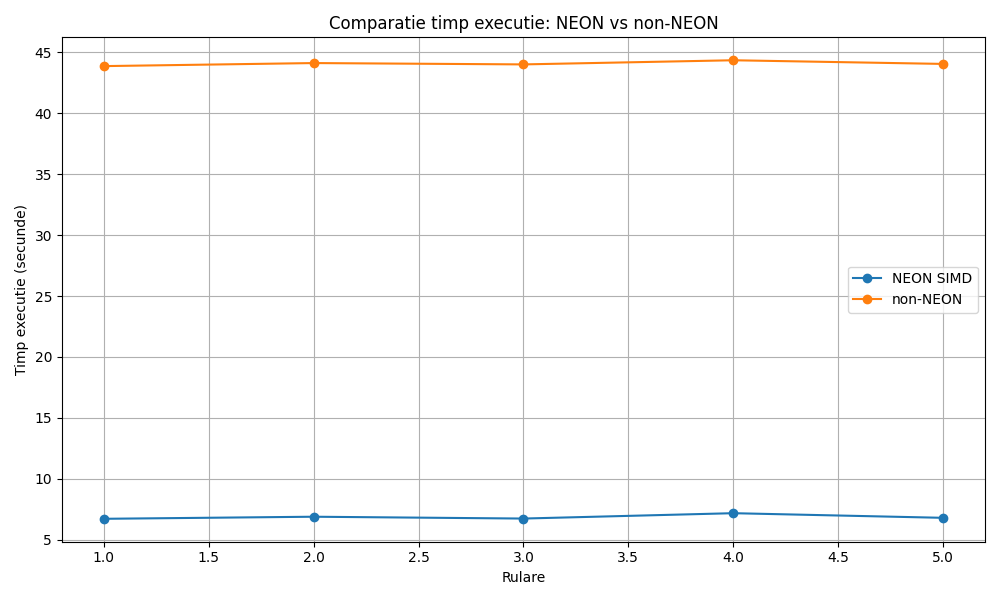
- Marcaje colorate pentru fiecare minutia detectata

- Output vizual salvat in: /home/user/PDS/neon/output\_images/result.png

- Output statistic, salveaza timpul de rulare intr-un fisier text: /home/user/PDS/neon/logs/result.txt

Folosind scripturi de Python si libraria Matplot, am generat diagrame care vizualizeaza rezultatele rularii celor 2 versiuni ale aplicatiei, punand in evidenta avantajele accelerarii codului sursa folosind instructiuni NEON:





# Concluzii

Utilizarea instructiunilor NEON a dus la o accelerare de peste 6 ori a algoritmului de extragere a minutiae-lor din amprente. Optimizarea este evidenta si justificata de natura paralelizabila a operatiilor pe imagine. Acest tip de accelerare este recomandat pentru sisteme embedded cu procesare intensiva pe imagini.

# Bibliografie

* Ruiz-Garcia, A., Dornaika, F., & Garcia-Zapirain, B. (2018). SOCOFing: Synthetic and Real Fingerprint Dataset for Testing Fingerprint Enhancement and Fingerprint Recognition Algorithms.

Sursă: <https://www.kaggle.com/datasets/ruizgara/socofing>

* Zhang, T. Y., & Suen, C. Y. (1984). A fast parallel algorithm for thinning digital patterns. Communications of the ACM, 27(3), 236–239.

Sursă: <https://doi.org/10.1145/357994.358023>

* OpenCV Documentation: <https://docs.opencv.org/4.x/index.html>

# Anexă Cod Sursă

## Varianta neoptimizată:

**main.cpp:**

#include <iostream>

#include <fstream>

#include <opencv2/opencv.hpp>

#include <vector>

#include <chrono>

#include <numeric>

// Functie care numara tranzitiile 0 -> 1 in jurul unui pixel

int count\_transitions**(**const std**::**vector**<**int**>&** neighbors**)** **{**

int transitions **=** 0**;**

**for** **(**size\_t k **=** 0**;** k **<** neighbors**.**size**();** k**++)** **{**

**if** **(**neighbors**[**k**]** **==** 0 **&&** neighbors**[(**k **+** 1**)** **%** neighbors**.**size**()]** **==** 1**)** **{**

transitions**++;**

**}**

**}**

**return** transitions**;**

**}**

// Functie care extrage cei 8 vecini ai unui pixel

std**::**vector**<**int**>** get\_neighbors**(**const cv**::**Mat**&** img**,** int i**,** int j**)** **{**

std**::**vector**<**int**>** neighbors **=** **{**

img**.**at**<**uchar**>(**i **-** 1**,** j**)** **>** 0 **?** 1 **:** 0**,**

img**.**at**<**uchar**>(**i **-** 1**,** j **+** 1**)** **>** 0 **?** 1 **:** 0**,**

img**.**at**<**uchar**>(**i**,** j **+** 1**)** **>** 0 **?** 1 **:** 0**,**

img**.**at**<**uchar**>(**i **+** 1**,** j **+** 1**)** **>** 0 **?** 1 **:** 0**,**

img**.**at**<**uchar**>(**i **+** 1**,** j**)** **>** 0 **?** 1 **:** 0**,**

img**.**at**<**uchar**>(**i **+** 1**,** j **-** 1**)** **>** 0 **?** 1 **:** 0**,**

img**.**at**<**uchar**>(**i**,** j **-** 1**)** **>** 0 **?** 1 **:** 0**,**

img**.**at**<**uchar**>(**i **-** 1**,** j **-** 1**)** **>** 0 **?** 1 **:** 0

**};**

**return** neighbors**;**

**}**

// Algoritmul Zhang-Suen pentru subtierea imaginii (fara NEON)

cv**::**Mat zhang\_suen\_thinning\_plain**(**const cv**::**Mat**&** binary\_image**,** int max\_iter **=** 20**)** **{**

cv**::**Mat thinned**;**

binary\_image**.**copyTo**(**thinned**);**

thinned **/=** 255**;** // Convertim imaginea binara la valori 0 si 1

int iteration **=** 0**;**

cv**::**Mat prev **=** cv**::**Mat**::**zeros**(**thinned**.**size**(),** CV\_8UC1**);**

**while** **(true)** **{**

cv**::**Mat marker **=** cv**::**Mat**::**zeros**(**thinned**.**size**(),** CV\_8UC1**);**

// Prima subetapa

**for** **(**int i **=** 1**;** i **<** thinned**.**rows **-** 1**;** i**++)** **{**

**for** **(**int j **=** 1**;** j **<** thinned**.**cols **-** 1**;** j**++)** **{**

**if** **(**thinned**.**at**<**uchar**>(**i**,** j**)** **!=** 1**)** **continue;**

std**::**vector**<**int**>** neighbors **=** get\_neighbors**(**thinned**,** i**,** j**);**

int count **=** std**::**accumulate**(**neighbors**.**begin**(),** neighbors**.**end**(),** 0**);**

int transitions **=** count\_transitions**(**neighbors**);**

**if** **(**count **>=** 2 **&&** count **<=** 6 **&&** transitions **==** 1 **&&**

neighbors**[**0**]** **\*** neighbors**[**2**]** **\*** neighbors**[**4**]** **==** 0 **&&**

neighbors**[**2**]** **\*** neighbors**[**4**]** **\*** neighbors**[**6**]** **==** 0**)** **{**

marker**.**at**<**uchar**>(**i**,** j**)** **=** 1**;**

**}**

**}**

**}**

thinned **-=** marker**;**

// A doua subetapa

marker **=** cv**::**Mat**::**zeros**(**thinned**.**size**(),** CV\_8UC1**);**

**for** **(**int i **=** 1**;** i **<** thinned**.**rows **-** 1**;** i**++)** **{**

**for** **(**int j **=** 1**;** j **<** thinned**.**cols **-** 1**;** j**++)** **{**

**if** **(**thinned**.**at**<**uchar**>(**i**,** j**)** **!=** 1**)** **continue;**

std**::**vector**<**int**>** neighbors **=** get\_neighbors**(**thinned**,** i**,** j**);**

int count **=** std**::**accumulate**(**neighbors**.**begin**(),** neighbors**.**end**(),** 0**);**

int transitions **=** count\_transitions**(**neighbors**);**

**if** **(**count **>=** 2 **&&** count **<=** 6 **&&** transitions **==** 1 **&&**

neighbors**[**0**]** **\*** neighbors**[**2**]** **\*** neighbors**[**6**]** **==** 0 **&&**

neighbors**[**0**]** **\*** neighbors**[**4**]** **\*** neighbors**[**6**]** **==** 0**)** **{**

marker**.**at**<**uchar**>(**i**,** j**)** **=** 1**;**

**}**

**}**

**}**

thinned **-=** marker**;**

// Verificam daca imaginea s-a modificat sau am atins numarul maxim de iteratii

**if** **(**cv**::**countNonZero**(**thinned **!=** prev**)** **==** 0 **||** iteration**++** **>=** max\_iter**)** **{**

**break;**

**}**

thinned**.**copyTo**(**prev**);**

**}**

thinned **\*=** 255**;** // Convertim imaginea inapoi la 0/255

**return** thinned**;**

**}**

// Structura pentru reprezentarea unei minutii (terminatie sau bifurcatie)

struct Minutia **{**

cv**::**Point position**;**

std**::**string type**;**

**};**

// Functie pentru extragerea minutiei dintr-o imagine subtire

std**::**vector**<**Minutia**>** extract\_minutiae\_plain**(**const cv**::**Mat**&** thinned**)** **{**

std**::**vector**<**Minutia**>** minutiae**;**

**for** **(**int i **=** 1**;** i **<** thinned**.**rows **-** 1**;** i**++)** **{**

**for** **(**int j **=** 1**;** j **<** thinned**.**cols **-** 1**;** j**++)** **{**

**if** **(**thinned**.**at**<**uchar**>(**i**,** j**)** **==** 255**)** **{**

std**::**vector**<**int**>** neighbors **=** get\_neighbors**(**thinned**,** i**,** j**);**

int transitions **=** count\_transitions**(**neighbors**);**

int sum\_neighbors **=** std**::**accumulate**(**neighbors**.**begin**(),** neighbors**.**end**(),** 0**);**

**if** **(**transitions **==** 1 **&&** sum\_neighbors **>=** 1**)** **{**

minutiae**.**push\_back**({**cv**::**Point**(**j**,** i**),** "ending"**});**

**}** **else** **if** **(**transitions **==** 3**)** **{**

minutiae**.**push\_back**({**cv**::**Point**(**j**,** i**),** "bifurcation"**});**

**}**

**}**

**}**

**}**

**return** minutiae**;**

**}**

// Functie care proceseaza imaginea de 500 de ori si masoara timpul total

void process\_plain**(**const cv**::**Mat**&** binary\_image**,** const std**::**string**&** log\_file\_path**,**

const std**::**string**&** output\_img\_path**)** **{**

**using** clock **=** std**::**chrono**::**high\_resolution\_clock**;**

cv**::**Mat last\_thinned**;**

std**::**vector**<**Minutia**>** last\_minutiae**;**

auto start **=** clock**::**now**();**

// Rulam algoritmul de 500 de ori

**for** **(**int k **=** 0**;** k **<** 500**;** **++**k**)** **{**

last\_thinned **=** zhang\_suen\_thinning\_plain**(**binary\_image**,** 20**);**

last\_minutiae **=** extract\_minutiae\_plain**(**last\_thinned**);**

**}**

auto end **=** clock**::**now**();**

std**::**chrono**::**duration**<**double**>** elapsed **=** end **-** start**;**

// Salvam timpul de rulare in fisier log

std**::**ofstream log\_file**(**log\_file\_path**,** std**::**ios**::**app**);**

**if** **(**log\_file**.**is\_open**())** **{**

log\_file **<<** "Execution time (plain Zhang-Suen + minutiae, 500 runs): "

**<<** elapsed**.**count**()** **<<** " seconds\n"**;**

log\_file**.**close**();**

**}** **else** **{**

std**::**cerr **<<** "Cannot open log file: " **<<** log\_file\_path **<<** std**::**endl**;**

**}**

// Desenam minutia pe imagine si o salvam

cv**::**Mat output\_img**;**

cv**::**cvtColor**(**last\_thinned**,** output\_img**,** cv**::**COLOR\_GRAY2BGR**);**

**for** **(**const auto**&** m **:** last\_minutiae**)** **{**

cv**::**Scalar color **=** **(**m**.**type **==** "ending"**)** **?** cv**::**Scalar**(**0**,** 255**,** 0**)** **:** cv**::**Scalar**(**0**,** 0**,** 255**);**

cv**::**circle**(**output\_img**,** m**.**position**,** 2**,** color**,** **-**1**);**

**}**

cv**::**imwrite**(**output\_img\_path**,** output\_img**);**

std**::**cout **<<** "Execution time (plain, 500 runs): " **<<** elapsed**.**count**()** **<<** " seconds\n"**;**

std**::**cout **<<** "Saved image to: " **<<** output\_img\_path **<<** "\n"**;**

**}**

int main**()** **{**

// Cale imagine intrare, iesire, log

std**::**string input\_path **=** "/home/user/PDS/non\_neon/input\_images/100\_\_M\_Left\_index\_finger.BMP"**;**

std**::**string output\_path **=** "/home/user/PDS/non\_neon/output\_images/result\_plain.png"**;**

std**::**string log\_path **=** "/home/user/PDS/non\_neon/logs/runtime\_log\_plain.txt"**;**

// Citim imaginea grayscale

cv**::**Mat img **=** cv**::**imread**(**input\_path**,** cv**::**IMREAD\_GRAYSCALE**);**

**if** **(**img**.**empty**())** **{**

std**::**cerr **<<** "Error loading image\n"**;**

**return** **-**1**;**

**}**

// Aplicam CLAHE pentru imbunatatirea contrastului

cv**::**Ptr**<**cv**::**CLAHE**>** clahe **=** cv**::**createCLAHE**(**2.0**,** cv**::**Size**(**8**,** 8**));**

cv**::**Mat enhanced**;**

clahe**->**apply**(**img**,** enhanced**);**

// Binarizam imaginea

cv**::**Mat binary**;**

cv**::**adaptiveThreshold**(**enhanced**,** binary**,** 255**,** cv**::**ADAPTIVE\_THRESH\_GAUSSIAN\_C**,**

cv**::**THRESH\_BINARY\_INV**,** 11**,** 2**);**

// Procesam imaginea si masuram timpul pentru 500 iteratii

process\_plain**(**binary**,** log\_path**,** output\_path**);**

**return** 0**;**

**}**

**CMakeLists.txt:**

**cmake\_minimum\_required**(**VERSION** 3.10)

**project**(FingerprintPlain)

**set**(CMAKE\_CXX\_STANDARD **17**)

**set**(CMAKE\_CXX\_STANDARD\_REQUIRED **ON**)

# Gaseste OpenCV

**find\_package**(OpenCV **REQUIRED**)

# Adauga sursa din subdirectorul `src`

**add\_executable**(FingerprintPlain src/main.cpp)

# Leaga OpenCV

**target\_link\_libraries**(FingerprintPlain ${OpenCV\_LIBS})

## Varianta optimizată:

**main.cpp:**

#include <iostream>

#include <fstream>

#include <opencv2/opencv.hpp>

#include <vector>

#include <chrono>

#include <string>

#include <numeric>

#include <arm\_neon.h> // NEON intrinseci

// Numara tranzitiile 0->1 in vectorul de vecini (folosit in Zhang-Suen)

int count\_transitions**(**const std**::**vector**<**int**>&** neighbors**)** **{**

int transitions **=** 0**;**

**for** **(**size\_t k **=** 0**;** k **<** neighbors**.**size**();** k**++)** **{**

**if** **(**neighbors**[**k**]** **==** 0 **&&** neighbors**[(**k **+** 1**)** **%** neighbors**.**size**()]** **==** 1**)** **{**

transitions**++;**

**}**

**}**

**return** transitions**;**

**}**

// Obtine cei 8 vecini folosind NEON intrinseci (accelereaza accesul la pixeli vecini)

std**::**vector**<**int**>** get\_neighbors\_neon**(**const cv**::**Mat**&** img**,** int i**,** int j**)** **{**

// Colectam pixelii vecini in ordinea corecta (in jurul pixelului curent)

uint8x8\_t neighbors\_raw **=** vcreate\_u8**(**

**((**uint64\_t**)**img**.**at**<**uchar**>(**i **-** 1**,** j**)** **<<** 0**)** **|**

**((**uint64\_t**)**img**.**at**<**uchar**>(**i **-** 1**,** j **+** 1**)** **<<** 8**)** **|**

**((**uint64\_t**)**img**.**at**<**uchar**>(**i**,** j **+** 1**)** **<<** 16**)** **|**

**((**uint64\_t**)**img**.**at**<**uchar**>(**i **+** 1**,** j **+** 1**)** **<<** 24**)|**

**((**uint64\_t**)**img**.**at**<**uchar**>(**i **+** 1**,** j**)** **<<** 32**)** **|**

**((**uint64\_t**)**img**.**at**<**uchar**>(**i **+** 1**,** j **-** 1**)** **<<** 40**)|**

**((**uint64\_t**)**img**.**at**<**uchar**>(**i**,** j **-** 1**)** **<<** 48**)** **|**

**((**uint64\_t**)**img**.**at**<**uchar**>(**i **-** 1**,** j **-** 1**)** **<<** 56**)**

**);**

// Prag de 0 pentru a obtine 0 sau 1 (pixels binari)

uint8x8\_t thresholded **=** vcgt\_u8**(**neighbors\_raw**,** vdup\_n\_u8**(**0**));**

std**::**vector**<**int**>** neighbors**(**8**);**

**for** **(**int k **=** 0**;** k **<** 8**;** k**++)** **{**

neighbors**[**k**]** **=** **(**thresholded**[**k**]** **!=** 0**)** **?** 1 **:** 0**;**

**}**

**return** neighbors**;**

**}**

// Zhang-Suen thinning - 500 iteratii (sau pana la stabilizare)

// Foloseste get\_neighbors\_neon pentru acces rapid la vecini (accelerare NEON)

cv**::**Mat zhang\_suen\_thinning**(**const cv**::**Mat**&** binary\_image**,** int max\_iter **=** 500**)** **{**

cv**::**Mat thinned**;**

binary\_image**.**copyTo**(**thinned**);**

thinned **/=** 255**;** // Convertim imaginea la valori binare 0/1

int iteration **=** 0**;**

cv**::**Mat prev **=** cv**::**Mat**::**zeros**(**thinned**.**size**(),** CV\_8UC1**);**

**while** **(true)** **{**

cv**::**Mat marker **=** cv**::**Mat**::**zeros**(**thinned**.**size**(),** CV\_8UC1**);**

// Sub-iteratia 1

**for** **(**int i **=** 1**;** i **<** thinned**.**rows **-** 1**;** i**++)** **{**

**for** **(**int j **=** 1**;** j **<** thinned**.**cols **-** 1**;** j**++)** **{**

**if** **(**thinned**.**at**<**uchar**>(**i**,** j**)** **!=** 1**)** **continue;**

std**::**vector**<**int**>** neighbors **=** get\_neighbors\_neon**(**thinned**,** i**,** j**);**

int count **=** std**::**accumulate**(**neighbors**.**begin**(),** neighbors**.**end**(),** 0**);**

int transitions **=** count\_transitions**(**neighbors**);**

**if** **(**count **>=** 2 **&&** count **<=** 6 **&&** transitions **==** 1 **&&**

neighbors**[**0**]** **\*** neighbors**[**2**]** **\*** neighbors**[**4**]** **==** 0 **&&**

neighbors**[**2**]** **\*** neighbors**[**4**]** **\*** neighbors**[**6**]** **==** 0**)** **{**

marker**.**at**<**uchar**>(**i**,** j**)** **=** 1**;**

**}**

**}**

**}**

thinned **-=** marker**;**

// Sub-iteratia 2

marker **=** cv**::**Mat**::**zeros**(**thinned**.**size**(),** CV\_8UC1**);**

**for** **(**int i **=** 1**;** i **<** thinned**.**rows **-** 1**;** i**++)** **{**

**for** **(**int j **=** 1**;** j **<** thinned**.**cols **-** 1**;** j**++)** **{**

**if** **(**thinned**.**at**<**uchar**>(**i**,** j**)** **!=** 1**)** **continue;**

std**::**vector**<**int**>** neighbors **=** get\_neighbors\_neon**(**thinned**,** i**,** j**);**

int count **=** std**::**accumulate**(**neighbors**.**begin**(),** neighbors**.**end**(),** 0**);**

int transitions **=** count\_transitions**(**neighbors**);**

**if** **(**count **>=** 2 **&&** count **<=** 6 **&&** transitions **==** 1 **&&**

neighbors**[**0**]** **\*** neighbors**[**2**]** **\*** neighbors**[**6**]** **==** 0 **&&**

neighbors**[**0**]** **\*** neighbors**[**4**]** **\*** neighbors**[**6**]** **==** 0**)** **{**

marker**.**at**<**uchar**>(**i**,** j**)** **=** 1**;**

**}**

**}**

**}**

thinned **-=** marker**;**

**if** **(**cv**::**countNonZero**(**thinned **!=** prev**)** **==** 0 **||** iteration**++** **>=** max\_iter**)** **{**

**break;**

**}**

thinned**.**copyTo**(**prev**);**

**}**

thinned **\*=** 255**;** // Reconversie la 0/255 pentru vizualizare

**return** thinned**;**

**}**

// Structura pentru minutii fingerprint

struct Minutia **{**

cv**::**Point position**;**

std**::**string type**;** // "ending" sau "bifurcation"

**};**

// Extrage minutii folosind vecinii NEON si algoritmul standard

std**::**vector**<**Minutia**>** extract\_minutiae**(**const cv**::**Mat**&** thinned**)** **{**

std**::**vector**<**Minutia**>** minutiae**;**

**for** **(**int i **=** 1**;** i **<** thinned**.**rows **-** 1**;** i**++)** **{**

**for** **(**int j **=** 1**;** j **<** thinned**.**cols **-** 1**;** j**++)** **{**

**if** **(**thinned**.**at**<**uchar**>(**i**,** j**)** **==** 255**)** **{**

std**::**vector**<**int**>** neighbors **=** get\_neighbors\_neon**(**thinned**,** i**,** j**);**

int transitions **=** count\_transitions**(**neighbors**);**

int sum\_neighbors **=** std**::**accumulate**(**neighbors**.**begin**(),** neighbors**.**end**(),** 0**);**

**if** **(**transitions **==** 1 **&&** sum\_neighbors **>=** 1**)** **{**

minutiae**.**push\_back**({**cv**::**Point**(**j**,** i**),** "ending"**});**

**}** **else** **if** **(**transitions **==** 3**)** **{**

minutiae**.**push\_back**({**cv**::**Point**(**j**,** i**),** "bifurcation"**});**

**}**

**}**

**}**

**}**

**return** minutiae**;**

**}**

// Functie care proceseaza si masoara timpul total pentru thinning + extragerea minutiei pe 500 iteratii

void process\_and\_measure**(**const cv**::**Mat**&** binary\_image**,** const std**::**string**&** log\_file\_path**,**

const std**::**string**&** output\_img\_path**)** **{**

**using** clock **=** std**::**chrono**::**high\_resolution\_clock**;**

cv**::**Mat last\_thinned**;**

std**::**vector**<**Minutia**>** last\_minutiae**;**

auto start **=** clock**::**now**();**

**for** **(**int k **=** 0**;** k **<** 500**;** **++**k**)** **{**

last\_thinned **=** zhang\_suen\_thinning**(**binary\_image**,** 20**);**

last\_minutiae **=** extract\_minutiae**(**last\_thinned**);**

**}**

auto end **=** clock**::**now**();**

std**::**chrono**::**duration**<**double**>** elapsed\_seconds **=** end **-** start**;**

std**::**ofstream log\_file**(**log\_file\_path**,** std**::**ios**::**app**);**

**if** **(**log\_file**.**is\_open**())** **{**

log\_file **<<** "Execution time (Zhang-Suen + minutiae, 500 runs): " **<<** elapsed\_seconds**.**count**()** **<<** " seconds\n"**;**

log\_file**.**close**();**

**}** **else** **{**

std**::**cerr **<<** "Cannot open log file: " **<<** log\_file\_path **<<** std**::**endl**;**

**}**

// Salvam doar ultima imagine procesata

cv**::**Mat minutiae\_img**;**

cv**::**cvtColor**(**last\_thinned**,** minutiae\_img**,** cv**::**COLOR\_GRAY2BGR**);**

**for** **(**const auto**&** m **:** last\_minutiae**)** **{**

cv**::**Scalar color **=** **(**m**.**type **==** "ending"**)** **?** cv**::**Scalar**(**0**,** 255**,** 0**)** **:** cv**::**Scalar**(**0**,** 0**,** 255**);**

cv**::**circle**(**minutiae\_img**,** m**.**position**,** 2**,** color**,** **-**1**);**

**}**

cv**::**imwrite**(**output\_img\_path**,** minutiae\_img**);**

std**::**cout **<<** "Thinning + minutiae extraction done in: " **<<** elapsed\_seconds**.**count**()** **<<** " seconds for 500 runs\n"**;**

std**::**cout **<<** "Result image saved to: " **<<** output\_img\_path **<<** std**::**endl**;**

std**::**cout **<<** "Log written to: " **<<** log\_file\_path **<<** std**::**endl**;**

**}**

int main**()** **{**

// Calea imaginilor si fisierelor

std**::**string input\_path **=** "/home/user/PDS/neon/input\_images/100\_\_M\_Left\_index\_finger.BMP"**;**

std**::**string output\_path **=** "/home/user/PDS/neon/output\_images/result.png"**;**

std**::**string log\_path **=** "/home/user/PDS/neon/logs/runtime\_log.txt"**;**

// Citim imaginea in grayscale

cv**::**Mat img **=** cv**::**imread**(**input\_path**,** cv**::**IMREAD\_GRAYSCALE**);**

**if** **(**img**.**empty**())** **{**

std**::**cerr **<<** "Error reading image\n"**;**

**return** **-**1**;**

**}**

// Preprocesare: CLAHE

cv**::**Ptr**<**cv**::**CLAHE**>** clahe **=** cv**::**createCLAHE**(**2.0**,** cv**::**Size**(**8**,** 8**));**

cv**::**Mat enhanced**;**

clahe**->**apply**(**img**,** enhanced**);**

// Binarizare adaptiva

cv**::**Mat binary**;**

cv**::**adaptiveThreshold**(**enhanced**,** binary**,** 255**,** cv**::**ADAPTIVE\_THRESH\_GAUSSIAN\_C**,**

cv**::**THRESH\_BINARY\_INV**,** 11**,** 2**);**

// Procesam si masuram timpul DOAR pentru thinning + extragerea minutiei

process\_and\_measure**(**binary**,** log\_path**,** output\_path**);**

**return** 0**;**

**}**

**CMakeLists.txt:**

**cmake\_minimum\_required**(**VERSION** 3.10)

**project**(FingerprintProcessing **VERSION** 1.0 LANGUAGES CXX)

**set**(CMAKE\_CXX\_STANDARD **17**)

**set**(CMAKE\_CXX\_STANDARD\_REQUIRED True)

# Detectam arhitectura

**if**(CMAKE\_SYSTEM\_PROCESSOR **MATCHES** "aarch64")

**message**(**STATUS** "Compilare pentru ARM64 cu NEON si OpenMP activat")

**add\_compile\_options**(

-O3

-march=armv8-a+simd

-ftree-vectorize

-funsafe-**math**-optimizations

-fopenmp

)

**set**(OPENMP\_LIB\_NAMES "gomp")

**set**(OPENMP\_FOUND TRUE)

# Find OpenMP

**find\_package**(OpenMP **REQUIRED**)

**else**()

**message**(**STATUS** "Compilare pentru arhitectura non-ARM64 cu optimizari native si OpenMP")

**add\_compile\_options**(-O3 -march=native -fopenmp)

**find\_package**(OpenMP **REQUIRED**)

**endif**()

# Cautam OpenCV

**find\_package**(OpenCV **REQUIRED**)

**add\_executable**(fingerprint\_processing src/main.cpp)

**target\_include\_directories**(fingerprint\_processing PRIVATE ${OpenCV\_INCLUDE\_DIRS})

**target\_link\_libraries**(fingerprint\_processing PRIVATE

${OpenCV\_LIBS}

OpenMP::OpenMP\_CXX

)

# Setam optiuni specifice targetului (dublu cu -march=armv8-a+simd pentru siguranta)

**if**(CMAKE\_SYSTEM\_PROCESSOR **MATCHES** "aarch64")

**target\_compile\_options**(fingerprint\_processing PRIVATE

-march=armv8-a+simd

-ftree-vectorize

-funsafe-**math**-optimizations

)

**endif**()