

LABORATORUL 05

# Accelerare hardware: același algoritm, de zece ori mai repede

Durată: 3 ore | Suport: Capitolul 9 | Platformă: Raspberry Pi 5 · ARM NEON | Carte capitolul de referință

| CUPRINS |                                |
|---------|--------------------------------|
| 1.      | Obiectivele lucrării           |
| 2.      | Ce construim                   |
| 3.      | De ce SIMD                     |
| 4.      | Pregătirea                     |
| 5.      | Cadrul de măsurare             |
| 6.      | Cele patru variante            |
| 7.      | Programul de comparație        |
| 8.      | Energia pe cadru               |
| 9.      | Cât se poate accelera, de fapt |
| 10.     | Sarcini de lucru               |
| 11.     | Provocare                      |
| 12.     | Întrebări de verificare        |
| 13.     | Resurse                        |

Când un algoritm este prea lent, primul reflex este să cerem un procesor mai rapid. De obicei nu e nevoie: procesorul pe care îl aveți deja poate prelucra șaisprezece valori printr-o singură instrucțiune, dar nu o face decât dacă i se cere explicit. În această lucrare scriem același algoritm în patru variante și măsurăm nu doar cât de repede rulează fiecare, ci și câtă energie consumă.

## 1 Obiectivele lucrării

- ☐ Înțelegerea modelului de calcul o instrucțiune – mai multe date (SIMD)
- ☐ Verificarea a ceea ce vectorizează compilatorul singur și a ceea ce nu poate vectoriza
- ☐ Scrierea de cod cu funcții intrinseci NEON și înțelegerea instrucțiunilor generate
- ☐ Măsurarea corectă a timpului de execuție, cu încălzire și repetări
- ☐ Calculul **energiei pe cadru** - și descoperirea că varianta rapidă este și cea economică
- ☐ Aplicarea legii lui Amdahl pentru a estima limita accelerării
- ☐ Situarea SIMD-ului între celelalte forme de accelerare: mai multe fire, GPU, FPGA, circuite dedicate

## 2 Ce construim

Alegem un algoritm simplu și util - **conversia unei imagini color în tonuri de gri** - și îl scriem în patru variante, de la cea mai naivă la cea mai optimizată. Le măsurăm pe toate, cu același set de date, pe aceeași plăcuță, apoi adăugăm măsurătoarea de putere de la lucrarea 1 și calculăm energia pe cadru.

| Varianta                   | Ce schimbă                                   | Efort                   |
|----------------------------|----------------------------------------------|-------------------------|
| 1. Scalară                 | o buclă obișnuită, un pixel pe iterație      | referința               |
| 2. Autovectorizată         | același cod, alți parametri de compilare     | zero - doar recompilare |
| 3. NEON explicit           | funcții intrinseci, 8 pixeli pe instrucțiune | mediu                   |
| 4. NEON + fire de execuție | toate cele patru nuclee lucrează simultan    | mic peste varianta 3    |

### Algoritmul

Ochiul uman nu percepe la fel cele trei culori: este mult mai sensibil la verde decât la albastru. Conversia standard ține cont de asta:

**FORMULA, ÎN ARITMETICĂ PE NUMERE ÎNTREGI**

$gri = (77 \cdot R + 150 \cdot G + 29 \cdot B) \gg 8$

Coeficienții 77, 150 și 29 sunt aproximările pe 8 biți ale ponderilor 0,299 / 0,587 / 0,114, iar suma lor este exact 256 - deci împărțirea finală devine o simplă deplasare cu 8 poziții. Nicio operație în virgulă mobilă, nicio împărțire. Aceasta nu este o simplificare didactică: este exact ce face orice bibliotecă de prelucrare a imaginilor.

**3 De ce SIMD**

Un procesor obișnuit prelucrează o valoare pe instrucțiune. Dar registrele lui au 128 de biți, iar un pixel are 8. Restul de 120 de biți stau nefolosiți.

**SIMD** - *Single Instruction, Multiple Data* - umple registrul cu șaisprezece valori de câte 8 biți și aplică aceeași operație tuturor, simultan. Aceeași instrucțiune, același timp de execuție, de șaisprezece ori mai mult lucru făcut.

**► PRELUCRARE SCALARĂ FAȚĂ DE PRELUCRARE PE BENZI**

operația aplicată:  $y[i] = (x[i] \cdot 77) \gg 8$

COD SCALAR - 0 VALOARE PE ITERAȚIE

|     |     |    |     |     |     |     |     |     |    |     |     |     |     |    |    |    |     |     |
|-----|-----|----|-----|-----|-----|-----|-----|-----|----|-----|-----|-----|-----|----|----|----|-----|-----|
| 11  | 48  | 85 | 122 | 159 | 196 | 233 | 14  | 51  | 88 | 125 | 162 | 199 | 236 | 17 | 54 | 91 | 128 | 165 |
| 202 | 239 | 20 | 57  | 94  | 131 | 168 | 205 | 242 | 23 | 60  | 97  | 134 |     |    |    |    |     |     |

NEON / ASIMD - 16 VALORI PE INSTRUȚIUNE

|     |     |    |     |     |     |     |     |     |    |     |     |     |     |    |    |    |     |     |
|-----|-----|----|-----|-----|-----|-----|-----|-----|----|-----|-----|-----|-----|----|----|----|-----|-----|
| 11  | 48  | 85 | 122 | 159 | 196 | 233 | 14  | 51  | 88 | 125 | 162 | 199 | 236 | 17 | 54 | 91 | 128 | 165 |
| 202 | 239 | 20 | 57  | 94  | 131 | 168 | 205 | 242 | 23 | 60  | 97  | 134 |     |    |    |    |     |     |

pași executați: 0

scalar: 0 / 32 valori gata - are nevoie de 32 iterații

NEON: 0 / 32 valori gata - are nevoie de 2 instrucțiuni

accelerare teoretică: 16x - în practică 6-12x, din cauza încărcărilor/stocărilor și a limitării de bandă a memoriei.

Versiune interactivă pe site.

**APĂSAȚI „RULEAZĂ” ȘI URMĂRIȚI DIFERENȚA**

Rândul de sus avansează cu o valoare pe pas. Rândul de jos avansează cu șaisprezece. Ambele fac exact același calcul, pe același procesor. Diferența este doar că a doua variantă folosește lățimea registrului care oricum există.

**Ce se poate și ce nu se poate vectoriza**

| Se vectorizează bine                            | Nu se poate vectoriza                              |
|-------------------------------------------------|----------------------------------------------------|
| aceeași operație pe multe elemente independente | iterația $i$ depinde de rezultatul iterației $i-1$ |

|                                             |                                                   |
|---------------------------------------------|---------------------------------------------------|
| acces secvențial la memorie                 | acces prin indirectare: <code>a[b[i]]</code>      |
| număr de iterații cunoscut înainte de buclă | bucla se oprește pe o condiție dependentă de date |
| ramificații care se pot exprima prin măști  | apeluri de funcții în corpul buclei               |

### DEPENDENȚA DE DATE ESTE BARIERA FUNDAMENTALĂ

nu se poate vectoriza - și niciun compilator nu o va face

```
for (int i = 1; i < n; i++)
    a[i] = a[i-1] * 0.9 + b[i] * 0.1;    // filtru recursiv
```

Fiecare rezultat are nevoie de cel precedent. Nu este o limitare a compilatorului, ci a algoritmului: calculele **NU SUNT INDEPENDENTE**

, deci nu pot fi făcute simultan. Când vă loviți de o buclă pe care nimic nu o accelerează, întrebați-vă întâi dacă nu cumva aceasta este cauza.

## 4 Pregătirea

1

### Verificați ce știe procesorul

```
$ capabilități

lscpu | grep -i "model name\|flags\|arhitect"
cat /proc/cpuinfo | grep -m1 Features
```

Pe Raspberry Pi 5 (Cortex-A76, ARMv8.2-A pe 64 de biți) trebuie să apară asimd - numele oficial al setului NEON pe 64 de biți. Pe arhitectura aarch64, NEON este **obligatoriu**, deci nu e nevoie de niciun parametru special de compilare pentru a-l activa.

2

### Creați folderul de lucru

```
$ folder

mkdir -p ~/si-lab/lab05 && cd ~/si-lab/lab05
```

3

### Fixați condițiile de măsurare

Fără acest pas, rezultatele variază de la o rulare la alta și comparațiile nu au valoare.

```
$ condiții stabile
```

```
# frecventa fixa la maximum, ca sa nu masuram guvernatorul in loc de algoritm
echo performance | sudo tee /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor

# fara interfata grafica
sudo systemctl isolate multi-user.target

# verificam ca nu exista limitari termice
vcgencmd measure_temp && vcgencmd get_throttled
```

### OBLIGATORIU

get\_throttled trebuie să întoarcă 0x0. Dacă procesorul este limitat termic, veți măsura eficiența răcitorului, nu a codului.

## 5 Cadrul de măsurare

Înainte de algoritm, avem nevoie de o metodă corectă de măsurare. Aceasta este partea pe care o greșesc cei mai mulți: o singură rulare a unei bucle scurte măsoară zgomot, nu performanță.

```
comun.h
```

```
#ifndef COMUN_H
#define COMUN_H

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

/* Imagine de test: 1920 x 1080, trei octeti pe pixel. */
#define LATIME 1920
#define INALTIME 1080
#define PIXELI ((size_t)(LATIME) * (INALTIME))

/* Ceas monoton: nu sare inapoi cand se sincronizeaza ora sistemului. */
static inline double acum(void) {
    struct timespec t;
    clock_gettime(CLOCK_MONOTONIC, &t);
    return t.tv_sec + t.tv_nsec * 1e-9;
}

/* Imagine determinista: aceleasi date la fiecare rulare, deci comparabile. */
static void umple(uint8_t* rgb, size_t pixeli) {
    for (size_t i = 0; i < pixeli; i++) {
        rgb[i * 3 + 0] = (uint8_t)((i * 7) & 0xFF);
        rgb[i * 3 + 1] = (uint8_t)((i * 13) & 0xFF);
        rgb[i * 3 + 2] = (uint8_t)((i * 29) & 0xFF);
    }
}

/* Verificarea corectitudinii: o varianta rapida dar gresita nu valoreaza nimic. */
static int compara(const uint8_t* a, const uint8_t* b, size_t n, const char* nume) {
    for (size_t i = 0; i < n; i++) {
```

```

        if (a[i] != b[i]) {
            printf("  EROARE in %s la pixelul %zu: %u != %u\n", nume, i, a[i], b[i]);
            return 0;
        }
    }
    return 1;
}

/* Masurare cu incalzire si repetari: intoarce timpul MEDIAN, nu media.
   Medianul nu este influentat de o rulare izolata perturbata de sistem. */
static int cmp_double(const void* x, const void* y) {
    double a = *(const double*)x, b = *(const double*)y;
    return (a > b) - (a < b);
}

#define REPETARI 11

static double masoara(void (*functie)(const uint8_t*, uint8_t*, size_t),
                      const uint8_t* intrare, uint8_t* iesire, size_t pixeli) {
    /* Incalzire: aduce datele in memoria cache si stabilizeaza frecventa. */
    for (int i = 0; i < 3; i++) functie(intrare, iesire, pixeli);

    double timpi[REPETARI];
    for (int i = 0; i < REPETARI; i++) {
        double t0 = acum();
        functie(intrare, iesire, pixeli);
        timpi[i] = acum() - t0;
    }
    qsort(timpi, REPETARI, sizeof(double), cmp_double);
    return timpi[REPETARI / 2];
}

#endif

```

## TREI DECIZII CARE FAC DIFERENȚA DINTRE O MĂSURĂTOARE ȘI O PĂRERE

### 1. ÎNCĂLZIREA

- prima rulare aduce datele în memoria cache și lasă procesorul să urce la frecvența maximă. Este sistematic mai lentă și trebuie aruncată.

### 2. MEDIANUL, NU MEDIA

- o singură rulare perturbată de sistemul de operare deplasează media, dar nu atinge medianul.

### 3. VERIFICAREA REZULTATULUI

- o variantă „optimizată” care dă alt rezultat nu este o optimizare, ci o eroare. Compară cu varianta de referință este obligatorie.

## 6 Cele patru variante

### Varianta 1 - scalară, referința

gri.c - varianta scalară

```
#include "comun.h"

void gri_scalar(const uint8_t* rgb, uint8_t* gri, size_t pixeli) {
    for (size_t i = 0; i < pixeli; i++) {
        uint32_t r = rgb[i * 3 + 0];
        uint32_t g = rgb[i * 3 + 1];
        uint32_t b = rgb[i * 3 + 2];
        gri[i] = (uint8_t)((77 * r + 150 * g + 29 * b) >> 8);
    }
}
```

### Varianta 2 - aceeași sursă, alți parametri de compilare

Compilerul poate vectoriza singur această buclă. Îi cerem să ne spună dacă a reușit:

\$ ce a vectorizat compilerul

```
gcc -O2 -c gri.c -o /dev/null -fopt-info-vec-optimized
echo "--- acum cu O3 ---"
gcc -O3 -mcpu=cortex-a76 -c gri.c -o /dev/null -fopt-info-vec-optimized
```

#### CITIȚI CU ATENȚIE IEȘIREA

Cu -O2, GCC nu vectorizează în mod implicit. Cu -O3 apare un mesaj de forma „loop vectorized using 16 byte vectors”. Dacă în locul lui apare -fopt-info-vec-missed cu o explicație,

#### ACOLO ESTE INFORMAȚIA VALOROASĂ

: compilerul vă spune exact ce l-a împiedicat. Cea mai frecventă cauză este că nu poate demonstra că zonele de intrare și de ieșire nu se suprapun în memorie.

#### CUVÂNTUL-CHEIE RESTRICT

Îi promite compilerului că doi pointeri nu indică spre aceeași zonă de memorie. Fără această promisiune, compilerul trebuie să presupună ce e mai rău și renunță la vectorizare:

o singură promisiune care schimbă codul generat

```
void gri_scalar(const uint8_t* restrict rgb, uint8_t* restrict gri, size_t pixeli)
```

Recompilați și comparați mesajele. Este una dintre puținele situații în care un singur cuvânt schimbă performanța cu zeci de procente.

### Varianta 3 - NEON explicit

gri\_neon.c

```

#include "comun.h"
#include <arm_neon.h>          /* functiile intrinseci NEON */

void gri_neon(const uint8_t* rgb, uint8_t* gri, size_t pixeli) {
    /* Ponderile, replicate in toate cele 8 benzi ale registrului. */
    const uint8x8_t p_r = vdup_n_u8(77);
    const uint8x8_t p_g = vdup_n_u8(150);
    const uint8x8_t p_b = vdup_n_u8(29);

    size_t i = 0;
    for (; i + 8 <= pixeli; i += 8) {
        /* vld3_u8 incarca 24 de octeti si ii DESPARTE automat pe cele trei
           canale: 8 valori R, 8 valori G, 8 valori B. Exact ce ne trebuie
           pentru date intretesute - o instructiune pe care un compilator
           rareori o alege singur. */
        uint8x8x3_t px = vld3_u8(rgb + i * 3);

        /* Inmultire cu largire: 8 biti x 8 biti -> 16 biti, ca sa nu depaseasca.
           77 * 255 = 19635, deci 8 biti nu ar fi ajuns. */
        uint16x8_t suma = vmull_u8(px.val[0], p_r);

        /* Inmultire cu largire SI acumulare, intr-o singura instructiune. */
        suma = vmlal_u8(suma, px.val[1], p_g);
        suma = vmlal_u8(suma, px.val[2], p_b);

        /* Deplasare la dreapta cu 8 SI ingustare inapoi la 8 biti,
           tot printr-o singura instructiune. */
        vst1_u8(gri + i, vshrn_n_u16(suma, 8));
    }

    /* Coadă: pixelii ramasi, cand numarul lor nu e multiplu de 8.
       Aceasta parte se uita foarte usor si produce o dunga gresita in imagine. */
    for (; i < pixeli; i++) {
        uint32_t r = rgb[i * 3 + 0], g = rgb[i * 3 + 1], b = rgb[i * 3 + 2];
        gri[i] = (uint8_t)((77 * r + 150 * g + 29 * b) >> 8);
    }
}

```

#### DE CE CÂȘTIGĂ VARIANTA SCRISĂ DE MÂNĂ

Trei instrucțiuni fac aici munca a zeci de instrucțiuni scalare:

- `vld3_u8` - desparte datele întrețesute R,G,B,R,G,B... în trei registre separate. Un compilator generează pentru asta o secvență lungă de permutări.
- `vmlal_u8` - înmulțește și acumulează într-o singură operație, cu lărgire la 16 biți.
- `vshrn_n_u16` - deplasează și îngustează simultan, în loc de două instrucțiuni.

Compilatorul nu are cum să știe că datele sunt întrețesute în grupuri de trei. Dumneavoastră știți.

## Varianta 4 - NEON pe toate nucleele

gri\_paralel.c

```
#include "comun.h"
#include <omp.h>

void gri_neon(const uint8_t* rgb, uint8_t* gri, size_t pixeli); /* varianta 3 */

void gri_paralel(const uint8_t* rgb, uint8_t* gri, size_t pixeli) {
    int fire = omp_get_max_threads();

    /* Impartim in bucati multiplu de 8, ca fiecare fir sa poata folosi
       calea NEON completa, fara coada proprie. */
    size_t bucata = ((pixeli / fire) / 8) * 8;

    #pragma omp parallel for schedule(static)
    for (int f = 0; f < fire; f++) {
        size_t start = f * bucata;
        size_t cate = (f == fire - 1) ? (pixeli - start) : bucata;
        gri_neon(rgb + start * 3, gri + start, cate);
    }
}
```

\$ compilare și rulare

```
gcc -O3 -mcpu=cortex-a76 -fopenmp \
    principal.c gri.c gri_neon.c gri_paralel.c -o comparatie
./comparatie
```

## 7 Programul de comparație

principal.c

```
#include "comun.h"

void gri_sclar(const uint8_t*, uint8_t*, size_t);
void gri_neon(const uint8_t*, uint8_t*, size_t);
void gri_paralel(const uint8_t*, uint8_t*, size_t);

typedef struct {
    const char* nume;
    void (*functie)(const uint8_t*, uint8_t*, size_t);
} Varianta;

int main(void) {
    uint8_t* rgb = malloc(PIXELI * 3);
    uint8_t* iesire = malloc(PIXELI);
    uint8_t* referinta = malloc(PIXELI);
    if (!rgb || !iesire || !referinta) return 1;

    umple(rgb, PIXELI);

    /* Referinta de corectitudine, calculata o singura data. */
    gri_sclar(rgb, referinta, PIXELI);
```

```

Varianta variante[] = {
    { "scalar",      gri_scalar },
    { "NEON",        gri_neon   },
    { "NEON + 4 fire", gri_paralel },
};

printf("image %d x %d = %.1f milioane de pixeli\n\n",
    LATIME, INALTIME, PIXELI / 1e6);
printf("%-16s %10s %12s %10s %8s\n",
    "variante", "timp", "Mpixeli/s", "accelerare", "corect");
printf("%s\n", "-----");

double t_referinta = 0;
for (size_t v = 0; v < sizeof(variante) / sizeof(variante[0]); v++) {
    memset(iesire, 0, PIXELI);
    double t = masoara(variante[v].functie, rgb, iesire, PIXELI);
    if (v == 0) t_referinta = t;

    int corect = compara(iesire, referinta, PIXELI, variante[v].nume);

    printf("%-16s %8.2f ms %12.1f %9.2fx %8s\n",
        variante[v].nume, t * 1000.0, PIXELI / t / 1e6,
        t_referinta / t, corect ? "da" : "NU");
}

/* Cate cadre pe secunda inseamna asta, in practica? */
printf("\nla 30 de cadre pe secunda, bugetul este de %.1f ms pe cadru\n",
    1000.0 / 30.0);

free(rgb); free(iesire); free(referinta);
return 0;
}

```

## Rezultate tipice

Valorile dumneavoastră vor diferi, dar raporturile ar trebui să fie asemănătoare:

| Varianta                     | Timp    | Mpixeli/s | Accelerare |
|------------------------------|---------|-----------|------------|
| scalar, -02                  | 18,4 ms | 113       | 1,00×      |
| scalar, -03 (autovectorizat) | 6,9 ms  | 301       | 2,67×      |
| NEON explicit                | 2,6 ms  | 798       | 7,08×      |
| NEON + 4 fire                | 0,9 ms  | 2305      | 20,4×      |

### DE CE NEON NU DĂ 8× EXACT

Prelucrăm 8 pixeli pe instrucțiune, deci ne-am aștepta la 8×. Obținem în jur de 7×, iar diferența are o explicație concretă:

### MEMORIA DEVINE STRANGULAREA

. O imagine de 1920×1080 în format RGB ocupă 6 MB - mai mult decât memoria cache a procesorului. La un moment dat, unitatea de calcul așteaptă datele, nu invers. De aceea varianta cu patru fire dă mai puțin de 4× peste NEON: cele patru nuclee împart aceeași magistrală către memorie. Aceasta este o limită fizică, nu un defect al codului.

## 8 Energia pe cadru

Aici lucrarea se leagă înapoi de prima. O variantă mai rapidă consumă o putere mai mare - dar pentru mai puțin timp. Care câștigă?

► energie\_cadru.py

```
#!/usr/bin/env python3
"""Masoara energia consumata de fiecare varianta a algoritmului."""

import subprocess
import sys
import threading
import time

sys.path.append("../lab01")          # reutilizam masuratoarea de la lucrarea 1
from putere_pmic import masoara

PERIOADA = 0.05                     # esantionare la 20 Hz
CADRE = 200                         # cate cadre prelucram pentru fiecare varianta

def energia_unei_rulari(comanda):
    """Ruleaza comanda si integreaza puterea pe toata durata ei."""
    probe = []
    opreste = threading.Event()

    def esantioneaza():
        while not opreste.is_set():
            p, _ = masoara()
            if p:
                probe.append((time.time(), p))
            time.sleep(PERIOADA)

    fir = threading.Thread(target=esantioneaza, daemon=True)
    fir.start()

    t0 = time.time()
    subprocess.run(comanda, stdout=subprocess.DEVNULL, check=True)
    durata = time.time() - t0

    opreste.set()
    fir.join(timeout=1)

    if len(probe) < 2:
        return durata, 0.0, 0.0

    # Integrare prin metoda trapezelor: mai exacta decat inmultirea cu media.
    energie = 0.0
    for (t1, p1), (t2, p2) in zip(probe, probe[1:]):
        energie += (p1 + p2) / 2 * (t2 - t1)

    putere_medie = energie / (probe[-1][0] - probe[0][0])
    return durata, putere_medie, energie

def main():
    print("Masor consumul in repaus (5 s)...")
    t0 = time.time()
    repaus = []
```

```

while time.time() - t0 < 5:
    p, _ = masoara()
    if p:
        repaus.append(p)
    time.sleep(PERIOADA)
p_repaus = sum(repaus) / len(repaus)
print("repaus: %.3f W\n" % p_repaus)

print("%-16s %9s %9s %10s %13s" %
      ("varianta", "durata", "putere", "energie", "mJ / cadru"))
print("-" * 62)

rezultate = []
for nume, indice in [("scalar", "0"), ("NEON", "1"), ("NEON + 4 fire", "2")]:
    durata, putere, energie = energia_unei_rulari(
        ["/comparatie_repetat", indice, str(CADRE)])
    pe_cadru = energie / CADRE * 1000 # milijouli
    util = (putere - p_repaus) * durata / CADRE * 1000
    rezultate.append((nume, durata, putere, energie, pe_cadru, util))
    print("%-16s %7.2f s %7.2f W %8.2f J %11.2f" %
          (nume, durata, putere, energie, pe_cadru))

cel_mai_econom = min(rezultate, key=lambda r: r[4])
referinta = rezultate[0][4]
print("\ncel mai econom: %s (%.1fx mai putina energie decat varianta scalara)"
      % (cel_mai_econom[0], referinta / cel_mai_econom[4]))

if __name__ == "__main__":
    main()

```

### REZULTATUL CARE LEAGĂ TOATĂ MATERIA

Varianta NEON consumă o putere

#### MAI MARE

- unitățile vectoriale comută mai multe tranzistoare pe ciclu. Dar rulează de șapte ori mai puțin timp. Energia pe cadru scade de aproximativ cinci ori.

Aceasta este exact strategia

#### „RACE TO IDLE”

din lucrarea 1, aplicată la nivel de instrucțiune în loc de nivel de frecvență. Nu am schimbat tensiunea, nu am schimbat frecvența - am folosit mai bine hardware-ul existent, și am obținut simultan viteză și economie. Este singura optimizare care nu presupune niciun compromis.

### CONSECINȚA PRACTICĂ PENTRU DISPOZITIVELE PE BATERIE

Un telefon care decodează un film cu instrucțiuni vectoriale nu o face pentru că altfel nu ar face față - o face pentru că altfel bateria ar ține jumătate. Aceași logică explică de ce codecurile video, criptografia și rețelele neuronale au, toate, implementări scrise de mână în asamblare vectorială.

## 9 Cât se poate accelera, de fapt

Am accelerat conversia de 20 de ori. Dar dacă ea reprezintă doar o parte din programul complet?

**LEGEA LUI AMDAHL**

accelerare totală =  $1 / ( (1 - p) + p/s )$

unde

**P**

este fracțiunea din timp ocupată de partea accelerată, iar

**S**

este de câte ori am accelerat-o.

| Cât din program accelerăm (p) | Accelerare locală s = 20× | Accelerare totală | Limita, chiar cu s = ∞ |
|-------------------------------|---------------------------|-------------------|------------------------|
| 50 %                          | 20×                       | 1,90×             | 2,00×                  |
| 80 %                          | 20×                       | 4,17×             | 5,00×                  |
| 95 %                          | 20×                       | 10,26×            | 20,00×                 |
| 99 %                          | 20×                       | 16,81×            | 100,00×                |

**CONSECINȚA PRACTICĂ**

Dacă partea pe care ați optimizat-o ocupă jumătate din timpul de execuție, nu veți depăși niciodată 2×, oricât de bine ați scrie codul.

**MĂSURĂȚI ÎNTÂI UNDE SE DUCE TIMPUL**

, apoi optimizați. Un profil obținut în cinci minute vă poate scuti de o săptămână de muncă în direcția greșită:

\$ unde se duce timpul

```
sudo apt install -y linux-perf
perf stat ./comparatie
perf record -g ./comparatie && perf report --stdio | head -30
```

**Locul SIMD-ului între celelalte forme de accelerare**

| Metodă                 | Accelerare tipică         | Efort       | Când se justifică                           |
|------------------------|---------------------------|-------------|---------------------------------------------|
| Algoritm mai bun       | nelimitată                | mare        | întotdeauna, prima dată                     |
| Parametri de compilare | 1,5–3×                    | minim       | întotdeauna, imediat după                   |
| SIMD scris de mână     | 4–8×                      | mediu       | bucle regulate, pe multe date               |
| Mai multe nuclee       | până la numărul de nuclee | mediu       | sarcini independente                        |
| GPU                    | 10–100×                   | mare        | volum mari, calcul masiv paralel            |
| FPGA                   | 10–1000×                  | foarte mare | serie medie, cerințe stricte de latență     |
| Circuit dedicat        | 100–10000×                | enorm       | serie foarte mare (codec-uri, criptografie) |

**ORDINEA NU ESTE ÎNTÂMPLĂTOARE**

Un algoritm mai bun bate orice accelerare hardware. O sortare de complexitate  $n \cdot \log n$  bate o sortare pătratică vectorizată, oricât de bine ar fi scrisă. SIMD este pasul care se face

**DUPĂ**

ce algoritmul este cel potrivit - nu în locul lui.

**CE OFERĂ RASPBERRY PI 5 ÎN PLUS**

Plăcuța are un procesor grafic VideoCore VII, utilizabil prin OpenGL ES sau Vulkan pentru calcul paralel, și un conector PCIe la care se poate atașa un accelerator de inteligență artificială. Nu are FPGA integrat, dar capitolul 9 din carte tratează arhitecturile reconfigurabile pe larg - iar principiul este același: mutați calculul repetitiv acolo unde poate fi făcut în paralel.

## 10 Sarcini de lucru

1. Compilați varianta scalară cu -O0, -O2 și -O3. Notați timpii și explicați diferența dintre ele.
2. Rulați -fopt-info-vec-missed pe varianta scalară fără restrict. Copiați în referat motivul exact invocat de compilator, apoi adăugați restrict și arătați cum se schimbă mesajul.
3. Rulați programul de comparație și completați tabelul cu propriile valori. Verificați coloana de corectitudine pentru fiecare variantă.
4. Ștergeți intenționat bucla de coadă din varianta NEON și rulați cu o imagine al cărei număr de pixeli nu este multiplu de 8. Descrieți ce se întâmplă și de ce eroarea este greu de observat vizual.
5. Măsurați energia pe cadru pentru toate cele trei variante. Realizați un grafic cu două axe: timp pe cadru și energie pe cadru.
6. Aplicați legea lui Amdahl: dacă în aplicația completă conversia ocupă 60 % din timp, ce accelerare totală obțineți cu varianta NEON + fire? Verificați rezultatul cu perf.
7. Modificați varianta NEON să prelucreze 16 pixeli pe iterație în loc de 8, folosind vld3q\_u8 și registrele complete de 128 de biți. Măsurați dacă mai câștigați ceva și explicați rezultatul prin limitarea de bandă a memoriei.

## 11 Provocare

### FILTRUL DE CONVOLUȚIE

Conversia în tonuri de gri este cel mai favorabil caz posibil: fiecare pixel se calculează independent, iar accesul la memorie este perfect secvențial. Încercați acum ceva mai greu - un

### FILTRU DE NETEZIRE 3×3

, în care fiecare pixel de ieșire depinde de cei nouă vecini ai săi.

Veți întâlni trei probleme reale, pe care conversia nu le are:

- **TREI RÂNDURI SIMULTAN.**

Trebuie citite trei linii ale imaginii deodată. Cum le păstrați în registre fără să reîncărcați aceleași date de trei ori?

- **MARGINILE.**

Ce se întâmplă la primul și la ultimul rând, unde vecinii lipsesc? Comparați cele două soluții obișnuite: prelucrarea separată a marginilor sau extinderea imaginii cu un chenar.

- **DEPĂȘIREA.**

Suma a nouă valori de 8 biți nu încapă pe 16 biți dacă ponderile sunt mari. Când trebuie să treceți la 32 de biți și cât vă costă asta în benzi pierdute?

Măsurați accelerarea obținută și comparați-o cu cea de la conversia în tonuri de gri. Explicați în referat de ce este mai mică - răspunsul spune tot ce trebuie știut despre limitele practice ale calculului vectorial.

## 12 Întrebări de verificare

### Ce înseamnă SIMD?

Mai multe procesoare rulează programe diferite

O singură instrucțiune se aplică simultan mai multor date

Instrucțiunile se execută în ordine aleatorie

Memoria este împărțită între nuclee

### De ce nu poate fi vectorizată bucla $a[i] = a[i-1] * 0.9 + b[i] * 0.1$ ?

Pentru că folosește numere în virgulă mobilă

Pentru că fiecare iterație depinde de rezultatul celei precedente

Pentru că vectorul este prea mic

Pentru că lipsește cuvântul restrict

**La ce folosește cuvântul-cheie restrict?**

Face variabila constantă

Promite compilatorului că doi pointeri nu indică spre aceeași zonă de memorie

Limitează domeniul de valori

Forțează alinierea în memorie

**Varianta NEON prelucrează 8 pixeli pe instrucțiune, dar accelerarea este de circa 7×, nu 8×. De ce?**

Din cauza buclei de coadă

Pentru că memoria devine strângut: imaginea nu încapă în memoria cache

Pentru că NEON rulează la frecvență mai mică

Din cauza erorilor de rotunjire

**De ce este obligatorie bucla de coadă în varianta NEON?**

Ca să golim registrele

Pentru că numărul de pixeli poate să nu fie multiplu de 8, iar ultimii ar rămâne neprelucrați

Ca să sincronizăm firele

Pentru compatibilitate cu procesoarele vechi

**Varianta NEON consumă mai multă putere. De ce consumă totuși mai puțină energie?**

Pentru că folosește mai puține nuclee

Pentru că rulează de aproximativ șapte ori mai puțin timp, iar  $E = P \cdot t$

Pentru că scade frecvența procesorului

Pentru că nu accesează memoria

**O parte care ocupă 50 % din timpul de execuție este accelerată de 20 de ori. Care este accelerarea totală?**

 10×

 1,90×

 20×

 2,50×

**Care este primul lucru de făcut când un program este prea lent?**

 Se scrie codul în asamblare vectorială

 Se măsoară unde se duce timpul

 Se cumpără un procesor mai rapid

 Se adaugă fire de execuție

**Răspunsuri:** 1-b, 2-b, 3-b, 4-b, 5-b, 6-b, 7-b, 8-b

1. Single Instruction, Multiple Data. Un registru de 128 de biți conține 16 valori de 8 biți, iar o instrucțiune le prelucerează pe toate deodată.
2. Este o dependență de date reală: iterațiile nu sunt independente, deci nu pot fi executate simultan. Nicio opțiune de compilare nu rezolvă asta - trebuie schimbat algoritmul.
3. Fără această promisiune, compilatorul trebuie să presupună că intrarea și ieșirea s-ar putea suprapune, caz în care vectorizarea ar da alt rezultat - deci renunță la ea.
4. O imagine RGB de 1920×1080 ocupă 6 MB, mai mult decât memoria cache. Unitatea vectorială ajunge să aștepte datele. Din același motiv, patru nuclee dau mai puțin de 4×.
5. Bucla vectorială avansează din 8 în 8. Dacă numărul total nu este multiplu de 8, ultimii pixeli rămân neatinși - o eroare care produce o dungă subțire, ușor de ratat vizual dar prezentă în rezultat.
6. Puterea crește pentru că sunt comutate mai multe tranzistoare pe ciclu, dar durata scade mult mai mult. Este strategia „race to idle” din lucrarea 1, aplicată la nivel de instrucțiune.
7. Legea lui Amdahl:  $1 / (0,5 + 0,5/20) = 1 / 0,525 \approx 1,90\times$ . Chiar cu accelerare infinită pe acea parte, limita ar fi 2×.
8. Fără profil, riscați să optimizați o parte care ocupă 5 % din timp. Legea lui Amdahl arată că efortul ar fi practic irosit. Măsurarea durează minute și direcționează toată munca următoare.

## 13 Resurse

Capitolul 9 - Sisteme reconfigurabile și accelerare hardware

ARM Intrinsics - căutarea tuturor funcțiilor NEON

Documentația procesoarelor Raspberry Pi

Capitolul 3 - Arhitectura memoriei și procesoarele ARM

GCC - opțiunile de optimizare și rapoartele de vectorizare

