



UNIVERSITÀ DELLA CALABRIA

DIPARTIMENTO DI  
INGEGNERIA INFORMATICA,  
MODELLISTICA, ELETTRONICA  
E SISTEMISTICA

DIMES

Corso di Laurea Magistrale in  
Ingegneria Informatica

Architetture Avanzate dei Sistemi di Elaborazione e  
Programmazione:  
“Correlation Feature Selection”  
in Linguaggio Assembly x86-32+SSE, x86-64+AVX e  
OpenMP

Gruppo 10

Prof. Fabrizio Angiulli	Desiré Chiappetta	257192
Prof. Fabio Fassetti	Antonio Paonessa	252318
Prof.ssa Simona Nisticò	Giulia Perri	242645

Anno Accademico 2023/2024



## Sommario

<b><i>Introduzione</i></b> .....	<b>4</b>
<b><i>Pseudocodice</i></b> .....	<b>5</b>
<b><i>Row-major Order vs Column-major Order</i></b> .....	<b>8</b>
Confronto Prestazioni Row-major Order e Column-major Order.....	12
<b><i>Ottimizzazioni</i></b> .....	<b>13</b>
1. Ottimizzazione con Loop Unrolling .....	13
2. Ottimizzazione con <i>media</i> , <i>rcf</i> e <i>rff</i> implementate in Assembly.....	15
2.1 MediaP4 .....	16
2.2 RcfP4 .....	19
2.2 RffP4 .....	20
Confronto Prestazioni Loop-Unroll in C e Code-Vectorization in Assembly .....	22
Confronto prestazioni col residuo .....	22
<b><i>Versione a 64 bit</i></b> .....	<b>23</b>
<b><i>Ottimizzazione con OpenMP</i></b> .....	<b>24</b>
<b><i>Confronto Prestazioni</i></b> .....	<b>26</b>
<b><i>Conclusione</i></b> .....	<b>27</b>

## Introduzione

La *feature selection* è il processo di selezione delle caratteristiche più interessanti per l'analisi di un insieme di dati, con l'obiettivo di ridurre la dimensionalità di un dataset e facilitarne l'estrazione di modelli e l'interpretabilità.

L'obiettivo del progetto è implementare un algoritmo di *feature selection* in linguaggio C e migliorarne le prestazioni, utilizzando tecniche di ottimizzazione tramite procedure in linguaggio Assembly x86-32 e x86-64 (con le estensioni SSE ed AVX).

Dato, quindi, un dataset definito su un insieme di features ed una variabile dicotomica in accordo alla quale l'insieme di dati viene suddiviso in due gruppi, l'algoritmo di *Correlation Feature Selection* usato individua un sottoinsieme  $S$  di features più rilevanti per l'analisi, ma allo stesso tempo non ridondanti tra di loro. A tal fine, utilizza il concetto di *correlazione*, scegliendo di volta in volta features altamente correlate con la variabile di classe ma scarsamente correlate tra di loro. L'analisi di correlazione può essere eseguita utilizzando misure statistiche come il coefficiente di correlazione di Pearson che misura la relazione lineare tra due variabili continue.

Sono state prodotte due diverse soluzioni: la prima ottimizzata per l'architettura x86-32+SSE, caratterizzata da registri a 32 bit e l'uso delle istruzioni SSE per le operazioni SIMD; la seconda, invece, è progettata per l'architettura x86-64+AVX, che supportando registri a 64 bit sfrutta le estensioni AVX per una maggiore parallelizzazione delle operazioni SIMD. Di entrambe le soluzioni, inoltre, sono state realizzate anche versioni che fanno uso dell'API OpenMP che contribuiscono a sfruttare al meglio le caratteristiche hardware delle CPU moderne.

L'obiettivo principale perseguito durante la realizzazione delle soluzioni, oltre alla correttezza dei risultati, è stato il miglioramento delle prestazioni; sono state, quindi, effettuate delle misurazioni con le diverse versioni realizzate, tramite la definizione di una funzione apposita, e i dati sono stati confrontati al fine di ridurre al minimo il tempo di esecuzione.

I test sono stati effettuati su:

- Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz,
- Cache size: 8192 KB,
- 4 core.

## Pseudocodice

Lo pseudocodice fornito è un'implementazione di un algoritmo di ricerca di un sottoinsieme ottimale di feature, mirando a massimizzare un "merito" basato su coefficienti di correlazione. Inizialmente, si esegue la ricerca della feature con rcf massimo.

```
S = {}
strcf = []
foreach f in features do
    Calcola rcf di f e memorizza risultato
    in store rcf (strcf)
    strcf[f] = rcf(f)

    confronta con rcf massimo
    if rcf > max(strcf) then
        fmax = f
    fi
end
```

Dopodiché, si aggiunge una feature con rcf massimo ad S e inizializza i valori iniziali di rcf e rff.

```
S = S U {fmax}
sumrcf = rcf[fmax]
sumrff = 0
while |S| < k do
    Ricerca della feature che aggiunta ad S dà punteggio massimo

    foreach fx in features do
        assegna a rcf_fx il valore di rcf di fx contenuto
        in strcf
        rcf_fx = strcf[fx]

        ricava rff per ogni coppia fx e fy con fy app. S
        e somma il risultato a rff_fx
        foreach fy in S do
            rff_fx += rff(fx, fy)
        end

        Aggiunge all'rcf e rff di S i valori ottenuti con la feature corrente
        rcf_curr = sumrcf + rcf_fx
        rff_curr = sumrff + rff_fx

        Calcola il valore medio del valore assoluto di tutte le correlazioni
        feature-classification
        midrcf = || rcf_curr || / (|S|+1)

        Calcola il valore medio del valore assoluto di tutte le correlazioni
        feature-feature rispetto a tutte le possibili
        coppie di feature in S U {fx}
        midrff = || rff_curr || / ( |S|*(|S|+1)/2 )

        Calcola il merit per S U {fx}
        merit = (|S|+1) * midrcf / sqrt(|S|+1 + (|S|+1) * |S| * midrff)
```

```

        Confronta merit massimo con merit corrente
        if merit < meritMax then
            La feature corrente diventa la possibile candidata ad entrare in S
            fmax = fx
            meritMax = merit
            rcfMax = rcf_curr
            rffMax = rff_curr
        fi
    end
    Aggiunge fmax all'insieme S e memorizza rcf e rff della soluzione
    corrente con |S| <= k
    S = S U {fmax}
    sumrcf = rcfMax
    sumrff = rffMax
end

```

La struttura principale è data da un ciclo che continua fino a quando la dimensione di  $S$  non raggiunge un valore prefissato  $k$ . All'interno del ciclo principale, c'è un secondo ciclo dedicato all'aggiunta iterativa delle feature all'insieme  $S$ . Per ogni feature aggiunta, vengono eseguite le seguenti operazioni:

- Calcolo del coefficiente di correlazione  $rcf$  basato sulla media e sulla deviazione standard.
- Calcolo del coefficiente di correlazione  $rff$  tra le feature presenti in  $S$ .
- Calcolo del "merito" basato su  $rcf$  e  $rff$ .
- Verifica se il merito calcolato è maggiore del massimo merito registrato  $meritMax$ .
- Se sì, l'insieme  $S$  viene aggiornato e il massimo merito viene registrato.

Lo pseudocodice utilizza diverse funzioni ausiliarie per semplificare le operazioni complesse e migliorare la leggibilità del codice:

- $rcf$ : Calcola il *point biserial correlation coefficient* rispetto alle classi 0 e 1.
- $rff$ : Calcola il *Pearson's correlation coefficient* tra due feature.
- $media$ : Calcola la media di una feature rispetto a una classe specifica e all'intero dataset.

Per testare il codice e valutarne le prestazioni, all'interno del main viene svolta la seguente funzione:

```

t = clock();
cfs(input);
t = clock() - t;
time = ((float)t)/CLOCKS_PER_SEC;

if(!input->silent)
    printf("CFS time = %.3f secs\n", time);
else
    printf("%.3f\n", time);

```

Inoltre, per facilitare lo svolgimento dei test abbiamo sviluppato del codice Python che automatizza la compilazione e l'esecuzione dei diversi programmi in ogni versione richiesta, implementando il salvataggio dei tempi su un file di testo.

```

import subprocess
import os
import time

def execute(cnt, file_name, cmd):
    with open(file_name, "w") as file:
        while cnt > 0:
            p = subprocess.run(cmd, shell=True, stdout=subprocess.PIPE, text=True)
            output = p.stdout.strip()
            file.write(f"{output}\n")
            cnt-=1

def buildC(cnt, version, path, dataset, labels, k, mode):
    os.system("gcc -o "+version+" "+version+".c -lm")
    execute(cnt, path+version+".txt", "./"+version+" -ds "+dataset+" -labels "+labels+" -k "+k+" -"+mode+"") 

def build32(cnt, version, path, dataset, labels, k, mode):
    os.system("nasm -f elf32 sseutils32.nasm && nasm -f elf32 cfs32.nasm")
    os.system("gcc -m32 -msse -O0 -no-pie sseutils32.o cfs32.o "+version+".c -o "+version+" -lm")
    execute(cnt, path+version+".txt", "./"+version+" -ds "+dataset+" -labels "+labels+" -k "+k+" -"+mode+"") 

def build64(cnt, version, path, dataset, labels, k, mode):
    os.system("nasm -f elf64 sseutils64.nasm && nasm -f elf64 cfs64.nasm")
    os.system("gcc -m64 -msse -mavx -O0 -no-pie sseutils64.o cfs64.o "+version+".c -o "+version+" -lm")
    execute(cnt, path+version+".txt", "./"+version+" -ds "+dataset+" -labels "+labels+" -k "+k+" -"+mode+"") 

def build32omp(cnt, version, path, dataset, labels, k, mode):
    os.system("nasm -f elf32 sseutils32.nasm && nasm -f elf32 cfs32.nasm")
    os.system("gcc -m32 -msse -O0 -no-pie -fopenmp sseutils32.o cfs32.o "+version+".c -o "+version+" -lm")
    execute(cnt, path+version+".txt", "./"+version+" -ds "+dataset+" -labels "+labels+" -k "+k+" -"+mode+"") 

def build64omp(cnt, version, path, dataset, labels, k, mode):
    os.system("nasm -f elf64 sseutils64.nasm && nasm -f elf64 cfs64.nasm")
    os.system("gcc -m64 -msse -mavx -O0 -no-pie -fopenmp sseutils64.o cfs64.o "+version+".c -o "+version+" -lm")
    execute(cnt, path+version+".txt", "./"+version+" -ds "+dataset+" -labels "+labels+" -k "+k+" -"+mode+"") 

```

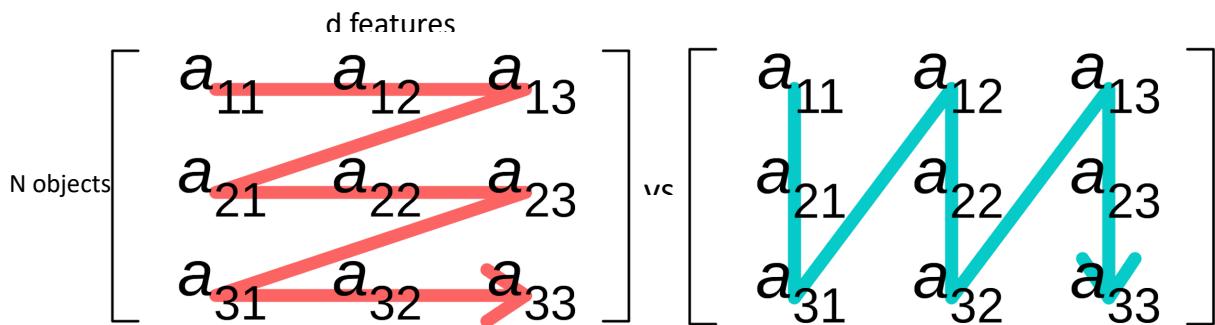
Dopo lo sviluppo dello pseudocodice, la risoluzione del problema si è sviluppata secondo due strategie:

- Row-major order;
- Column-major order;

In base alla modalità di caricamento del dataset, quindi, possiamo implementare la soluzione in due modi diversi per valutarne le prestazioni.

## Row-major Order vs Column-major Order

Nella versione Row-major order la matrice che rappresenta il dataset viene memorizzata in modo lineare con gli elementi di ogni riga contigui in memoria. Invece, nella versione Column-major order la matrice che rappresenta il dataset viene memorizzata in modo lineare, ma, rispetto al caso precedente, sono gli elementi di ogni colonna ad essere contigui in memoria.



Le funzioni principali del codice sono:

- *load\_data* e *load\_data\_column*: Funzioni per caricare i dati da un file binario. I dati sono memorizzati come una matrice.

```
MATRIX load_data(char* filename, int *n, int *k) {
    FILE* fp;
    int rows, cols, status, i;

    fp = fopen(filename, "rb");

    if (fp == NULL){
        printf("%s: bad data file name!\n", filename);
        exit(0);
    }

    status = fread(&cols, sizeof(int), 1, fp);
    status = fread(&rows, sizeof(int), 1, fp);

    MATRIX data = alloc_matrix(rows,cols);
    status = fread(data, sizeof(type), rows*cols, fp);
    fclose(fp);

    *n = rows;
    *k = cols;

    return data;
}

MATRIX load_data_column(char* filename, int *n, int *k) {
    FILE* fp;
    int rows, cols, status, i;

    fp = fopen(filename, "rb");

    if (fp == NULL){
        printf("%s: bad data file name!\n", filename);
        exit(0);
    }

    status = fread(&cols, sizeof(int), 1, fp);
    status = fread(&rows, sizeof(int), 1, fp);

    MATRIX data = alloc_matrix(rows,cols);
    status = fread(data, sizeof(type), rows*cols, fp);
    fclose(fp);

    *n = rows;
    *k = cols;

    MATRIX corder = alloc_matrix(cols,rows);
    for(int j=0; j<cols; j++)
        for(int i=0; i<rows; i++)
            corder[j*rows+i] = data[i*cols+j];

    dealloc_matrix(data);

    return corder;
}
```

- *count*: Ricava il numero di elementi del dataset appartenenti al gruppo 0.

```
int count(params* input){
    int cnt=0;
    for(int i=0; i<input->N; i++)
        if(input->labels[i]==0)
            cnt++;
    return cnt;
}
```

- *media*: Calcola le medie dei valori assunti sulla feature dal gruppo 0, dal gruppo 1 e complessivamente da tutti gli elementi.

```
void media(int fx, int n0, int n1, params* input, MATRIX mu) {
    type m=0.0, m0=0.0, m1=0.0;
    for(int i=0; i<input->N; i++) {
        if(input->labels[i]==0)
            m0+=input->ds[i*input->d+fx]/n0;
        else
            m1+=input->ds[i*input->d+fx]/n1;
        m+=input->ds[i*input->d+fx]/input->N;
    }
    mu[0*input->d+fx]=m0;
    mu[1*input->d+fx]=m1;
    mu[2*input->d+fx]=m;
}

void media(int fx, int n0, int n1, params* input, MATRIX mu) {
    type m=0.0, m0=0.0, m1=0.0;
    for(int i=0; i<input->N; i++) {
        if(input->labels[i]==0)
            m0+=input->ds[fx*input->N+i]/n0;
        else
            m1+=input->ds[fx*input->N+i]/n1;
        m+=input->ds[fx*input->N+i]/input->N;
    }
    mu[0*input->d+fx]=m0;
    mu[1*input->d+fx]=m1;
    mu[2*input->d+fx]=m;
}
```

- *rcf*: Calcola il valore del Relevance Correlation Feature (RCF) per una particolare feature.

```
void rcf(int fx, type cost1, type cost2, MATRIX mu, params* input, MATRIX strcf) {
    type mu0 = mu[0*input->d+fx];
    type mu1 = mu[1*input->d+fx];
    type muu = mu[2*input->d+fx];
    type ximu0 = 0.0;

    for(int i=0; i<input->N; i++)
        ximu0+=(input->ds[i*input->d+fx]-muu)*(input->ds[i*input->d+fx]-muu);

    type sigmaf = cost1 * sqrt(ximu0);
    strcf[fx] = fabs(mu0-mu1)* cost2/sigmaf;
}

void rcf(int fx, type cost1, type cost2, MATRIX mu, params* input, MATRIX strcf) {
    type mu0 = mu[0*input->d+fx];
    type mu1 = mu[1*input->d+fx];
    type muu = mu[2*input->d+fx];
    type ximu0 = 0.0;

    for(int i=0; i<input->N; i++)
        ximu0+=(input->ds[fx*input->N+i]-muu)*(input->ds[fx*input->N+i]-muu);

    type sigmaf = cost1 * sqrt(ximu0);
    strcf[fx] = fabs(mu0-mu1)* cost2/sigmaf;
}
```

- *rff*: Calcola la similarità tra due feature utilizzando la formula di correlazione.

```
type rff(int fx, int fy, MATRIX mu, params* input) {
    type mux = mu[2*input->d+fx];
    type muy = mu[2*input->d+fy];

    type num = 0.0, den1 = 0.0, den2 = 0.0;
    type ximux, yimuy;

    for(int i=0; i<input->N; i++){
        ximux=input->ds[fx*input->N+i]-mux;
        yimuy=input->ds[fy*input->N+i]-muy;

        num += ximux*yimuy;
        den1 += ximux*ximux;
        den2 += yimuy*yimuy;
    }

    return fabs(num) / sqrt(den1*den2);
}

type rff(int fx, int fy, MATRIX mu, params* input) {
    type mux = mu[2*input->d+fx];
    type muy = mu[2*input->d+fy];

    type num = 0.0, den1 = 0.0, den2 = 0.0;
    type ximux, yimuy;

    for(int i=0; i<input->N; i++){
        ximux=input->ds[i*input->d+fx]-mux;
        yimuy=input->ds[i*input->d+fy]-muy;

        num += ximux*yimuy;
        den1 += ximux*ximux;
        den2 += yimuy*yimuy;
    }

    return fabs(num) / sqrt(den1*den2);
}
```

- *cfs*: Implementa l'algoritmo CFS, selezionando iterativamente le features in base al massimo valore di RCF e ad un merito calcolato.

```
void cfs(params* input) {
    MATRIX mu = alloc_matrix(3, input->d);
    MATRIX strcf = alloc_matrix(1, input->d);
    int* inS = (int *)calloc(input->d, sizeof(int));
```

Sono state allocate alcune strutture dati di supporto all'algoritmo.

In particolare, *mu* è una matrice 3xd in cui vengono memorizzati i valori calcolati dalla funzione *media*: nella riga 0 vengono memorizzate le medie dei valori assunti dal gruppo 0 sulle feature, nella riga 1 le medie dei valori assunti dal gruppo 1 e nella riga 3 i valori medi complessivi.

Nell'array *strcf* vengono memorizzati i valori di rcf relativi ad ogni feature, in modo da non doverli calcolare più di una volta.

L'array *inS*, infine, serve a tenere traccia delle feature già inserite nella soluzione, così da non ricon siderarle nelle iterazioni successive.

```
int n0 = count(input), n1 = input->N - n0;
int fi;
for(fi=0; fi<input->d; fi++)
    media(fi, n0, n1, input, mu);

type cost1 = sqrt(1.0/(input->N-1)); // sqrt(1/N-1)
type cost2 = sqrt((n0*n1))/input->N; // sqrt(n0*n1) / N

// S=0
// cerco la feature con rcf max
int cardS=0;
type rcfMax=0.0;
int fmax=0;
for( fi=0; fi<input->d; fi++) {
    rcf(fi, cost1, cost2, mu, input, strcf);
    if (rcfMax < strcf[fi]) {
        rcfMax = strcf[fi];
        fmax = fi;
    }
}

input->out[cardS] = fmax;
inS[fmax]=1;
cardS++;
// S = {f : rcf max} e |S| = 1
```

```

type midrcfIn = strcf[fmax], midrffIn = 0.0;
type midrcf, midrff, midrcfMax, midrffMax;
type merit, meritMax;
while(cardS<input->k) {
    meritMax=0.0;
    for(int fx=0; fx<input->d; fx++) {
        if(inS[fx]==0){
            midrcf = midrcfIn + strcf[fx];
            midrff = midrffIn;

            for (int y = 0; y < cardS; y++)
                midrff += rff(fx, input->out[y], mu, input);

            merit = midrcf / sqrt((cardS+1) + midrff*2);
            if (meritMax < merit) {
                meritMax = merit;
                fmax = fx;
                midrcfMax = midrcf;
                midrffMax = midrff;
            }
        }
    }
    input->out[cardS] = fmax;
    inS[fmax]=1;
    cardS++;
    midrcfIn=midrcfMax;
    midrffIn=midrffMax;
    // S U {f : merit max } e |S| = 2,...,k
}
input->sc = meritMax;
dealloc_matrix(mu);
dealloc_matrix(strcf);
free(inS);
}

```

## Confronto Prestazioni Row-major Order e Column-major Order

La versione Column-major order garantisce che gli elementi di una stessa colonna siano contigui in memoria. Questo consente di minimizzare i costi associati ai cache miss, in quanto favorisce il principio di località spaziale. I test effettuati rappresentano proprio come questo approccio organizzativo contribuisce ad ottimizzare le prestazioni.

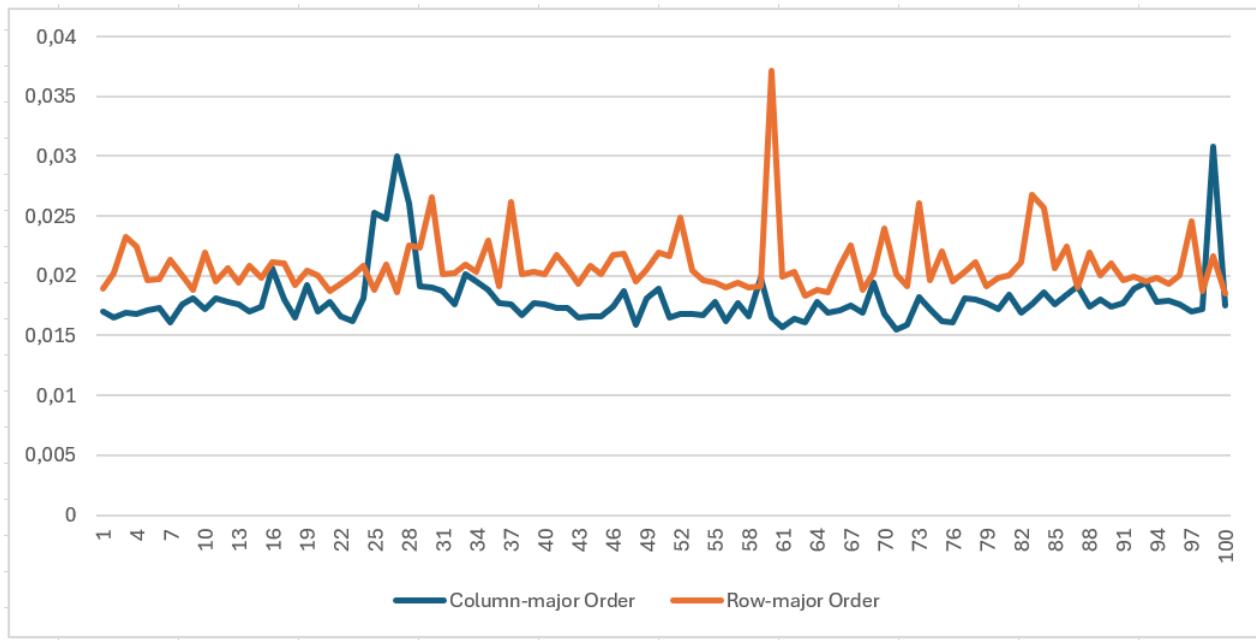


Figura 1: Confronto tra Row-major Order e Column-major Order

# Ottimizzazioni

## 1. Ottimizzazione con Loop Unrolling

La prima ottimizzazione introdotta a partire dalla versione Column-major order è stato il *loop unrolling* sui principali cicli for utilizzati nel programma. Il loop unrolling è importante in quanto permette di diminuire i costi legati alla gestione del ciclo.

In particolare, i cicli ottimizzati riguardano le funzioni *media*, *rcf* e *rff*. Il meccanismo prevede di gestire in un ciclo principale i primi  $N - N \% p$  elementi e poi di gestire in un secondo ciclo i restanti  $N \% p$  elementi di residuo del dataset.

Per tutte le funzioni è stato usato un fattore di unroll pari a 4.

```
void media(int fx, int n0, int n1, params* input, MATRIX mu) {
    type m=0.0, m0=0.0, m1=0.0;
    int i=0;
    for( ; i<input->N-input->N%4; i+=4) { // N-N%p
        if(input->labels[i]==0)
            m0+=input->ds[fx*input->N+i]/n0;
        else
            m1+=input->ds[fx*input->N+i]/n1;

        if(input->labels[i+1]==0)
            m0+=input->ds[fx*input->N+i+1]/n0;
        else
            m1+=input->ds[fx*input->N+i+1]/n1;

        if(input->labels[i+2]==0)
            m0+=input->ds[fx*input->N+i+2]/n0;
        else
            m1+=input->ds[fx*input->N+i+2]/n1;

        if(input->labels[i+3]==0)
            m0+=input->ds[fx*input->N+i+3]/n0;
        else
            m1+=input->ds[fx*input->N+i+3]/n1;

        m+=input->ds[fx*input->N+i]/input->N;
        m+=input->ds[fx*input->N+i+1]/input->N;
        m+=input->ds[fx*input->N+i+2]/input->N;
        m+=input->ds[fx*input->N+i+3]/input->N;
    }

    for( ; i<input->N; i++) { // residuo N%p
        if(input->labels[i]==0)
            m0+=input->ds[fx*input->N+i]/n0;
        else
            m1+=input->ds[fx*input->N+i]/n1;
        m+=input->ds[fx*input->N+i]/input->N;
    }

    mu[0*input->d+fx]=m0;
    mu[1*input->d+fx]=m1;
    mu[2*input->d+fx]=m;
}
```

```

void rcf(int fx, type cost1, type cost2, MATRIX mu, params* input, MATRIX strcf) {
    type mu0 = mu[0*input->d+fx];
    type mu1 = mu[1*input->d+fx];
    type muu = mu[2*input->d+fx];
    type ximuu = 0.0;
    int i=0;
    for( ; i<input->N-input->N%4; i+=4) { // N-N%p
        ximuu+=
            (input->ds[fx*input->N+i]-muu)*(input->ds[fx*input->N+i]-muu)+
            (input->ds[fx*input->N+i+1]-muu)*(input->ds[fx*input->N+i+1]-muu)+
            (input->ds[fx*input->N+i+2]-muu)*(input->ds[fx*input->N+i+2]-muu)+
            (input->ds[fx*input->N+i+3]-muu)*(input->ds[fx*input->N+i+3]-muu);
    }

    // residue N%p
    for( ; i<input->N; i++) {
        ximuu+=(input->ds[fx*input->N+i]-muu)*(input->ds[fx*input->N+i]-muu);
    }

    type sigmaf = cost1 * sqrt(ximuu);

    strcf[fx] = fabs(mu0-mu1)* cost2/sigmaf;
}

type rff(int fx, int fy, MATRIX mu, params* input) {
    type mux = mu[2*input->d+fx];
    type muy = mu[2*input->d+fy];
    type num = 0.0, den1 = 0.0, den2 = 0.0;
    type ximux, yimuy, ximux1, yimuy1, ximux2, yimuy2, ximux3, yimuy3;
    int i=0;
    for( ; i<input->N-input->N%4; i+=4){ // N-N%p
        ximux=input->ds[fx*input->N+i]-mux; // xi - mux
        yimuy=input->ds[fy*input->N+i]-muy; // yi - muy

        ximux1=input->ds[fx*input->N+i+1]-mux; // xi - mux
        yimuy1=input->ds[fy*input->N+i+1]-muy; // yi - muy

        ximux2=input->ds[fx*input->N+i+2]-mux; // xi - mux
        yimuy2=input->ds[fy*input->N+i+2]-muy; // yi - muy

        ximux3=input->ds[fx*input->N+i+3]-mux; // xi - mux
        yimuy3=input->ds[fy*input->N+i+3]-muy; // yi - muy

        num += ximux*yimuy + ximux1*yimuy1 + ximux2*yimuy2 + ximux3*yimuy3; // (xi - mux)*(yi - muy)
        den1 += ximux*ximux + ximux1*ximux1 + ximux2*ximux2 + ximux3*ximux3; // (xi - mux)^2
        den2 += yimuy*yimuy + yimuy1*yimuy1 + yimuy2*yimuy2 + yimuy3*yimuy3; // (yi - muy)^2
    }

    for( ; i<input->N; i++){ // residuo
        ximux=input->ds[fx*input->N+i]-mux; // xi - mux
        yimuy=input->ds[fy*input->N+i]-muy; // yi - muy

        num += ximux*yimuy; // (xi - mux)*(yi - muy)
        den1 += ximux*ximux; // (xi - mux)^2
        den2 += yimuy*yimuy; // (yi - muy)^2
    }

    return fabs(num) / sqrt(den1*den2);
}

```

Per il calcolo di rcf, il loop unrolling è stato definito sia per il ciclo sugli elementi, come appena descritto, sia per il ciclo sulle features, in questo caso con un fattore di unroll pari a 2.

```

// S=0
// cerco la feature con rcf max
int cardS=0;
type rcfMax=0.0;
int fmax=0;
fi=0;
for( ; fi<input->d-input->d%2; fi+=2) { // d-d%p
    rcf(fi, cost1, cost2, mu, input, strcf);
    rcf(fi+1, cost1, cost2, mu, input, strcf);

    if (strcf[fi] > strcf[fi+1]) {
        if(strcf[fi] > rcfMax) {
            rcfMax = strcf[fi];
            fmax = fi;
        }
    }
    else if(strcf[fi+1] > rcfMax) {
        rcfMax = strcf[fi+1];
        fmax = fi+1;
    }
}

// residuo d%p
for( ; fi<input->d; fi++) {
    rcf(fi, cost1, cost2, mu, input, strcf);

    if (rcfMax < strcf[fi]) {
        rcfMax = strcf[fi];
        fmax = fi;
    }
}

```

## 2. Ottimizzazione con *media*, *rcf* e *rff* implementate in Assembly

Il passo successivo è stato implementare le funzioni *media* e *rcf* in Assembly. L'utilizzo dei registri per il calcolo a virgola mobile (xmm0, ..., xmm7 per la versione a 32 bit, ymm0, ..., ymm15 per la versione a 64 bit) ha permesso di utilizzare la *code vectorization*, lavorando su quattro elementi del dataset contemporaneamente e riducendo le somme parziali ad un unico risultato con l'istruzione *hadd* al termine del ciclo.

Una parte del calcolo di *rcf* è stata integrata in *media*, così da evitare di passare molti parametri a *rcf*. Nello specifico, nella procedura *media* viene calcolata la quantità:

$$|\mu_0 - \mu_1| * \sqrt{\frac{n_0 * n_1}{N^2}} * \sqrt{N - 1}$$

E viene memorizzata all'interno di *strcf*. In questo modo in *rcf* è sufficiente dividere il risultato già memorizzato nell'array con la quantità:

$$\sqrt{\sum_{i=1}^N (x_i - \mu)^2}$$

In entrambe le procedure al termine dei cicli è stato gestito un eventuale residuo.

```
extern void mediaP4(int fx, int nmodp, MATRIX mu, params* input, MATRIX strcf);

extern int rcfP4(int dmodp, int nmodp, MATRIX mu, params* input, MATRIX strcf);

extern type rffP4(int fx, int fy, int nmodp, MATRIX mu, params* input);
```

Di seguito vengono riportati degli estratti del codice Assembly.

## 2.1 MediaP4

Nella procedura mediaP4 è stata sviluppata la tecnica di loop unrolling implementata in C, in combinazione con la tecnica di code vectorization.

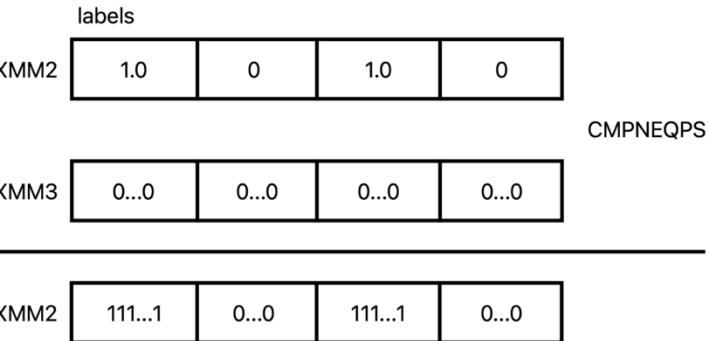
```
for_media:
    cmp esi, edx          ; i < N-N%p
    jnl res_media
    movups xmm0, [eax + ebx*4] ; [ ds[fx*N+i] | ds[fx*N+i+1] | ds[fx*N+i+2] | ds[fx*N+i+3] ]
    movups xmm1, xmm0        ; [ ds[fx*N+i] | ds[fx*N+i+1] | ds[fx*N+i+2] | ds[fx*N+i+3] ]
    movups xmm2, [ecx + esi*4]; [ labels[i] | labels[i+1] | labels[i+2] | labels[i+3] ]
```

A differenza di C, la gestione degli elementi appartenenti alle due classi (categoria=0 e categoria=1) avviene sfruttando il repertorio SSE. Nel registro xmm0 e xmm2 vengono allocati rispettivamente i 4 elementi del dataset e le corrispettive etichette.

```
addps xmm5, xmm2          ; counter n1
cmpneqps xmm2, xmm3       ; labels_mask based on c=1
```

La funzione *count* viene implementata nella procedura con un meccanismo basato sulla somma delle etichette di categoria, in questo modo a fine iterazione in xmm5 è presente il numero di oggetti di categoria=1.

L'istruzione *cmpneqs* viene utilizzata per costruire una maschera usando le etichette, in modo tale da poter considerare in modo separato gli oggetti di categoria 1 e di categoria 0.



```

andps xmm0, xmm2          ; ds and labels_mask      (c=1)
andps xmm2, xmm1          ; ds and not labels_mask (c=0)

addps xmm6, xmm0          ; m1 m1 m1 m1
addps xmm7, xmm2          ; m0 m0 m0 m0

add esi, 4                ; i+=4
add ebx, 4                ; (fx*N+i)+4
jmp for_media

```

La maschera viene confrontata con i registri contenenti gli oggetti della iterazione corrente e il risultato aggiunto ai registri accumulatore xmm6 e xmm7.

Per la gestione del residuo, si è definito un meccanismo basato sul considerare gli ultimi p oggetti relativi alla feature corrente per poi scartare gli elementi che non fanno parte del residuo.

L'idea è quella di definire una maschera basata sul valore del residuo ( $N\%4$ ), che per un fattore di unroll pari a 4, può valere {0, 1, 2, 3}.

```

res_media:
    mov edx, [ebp + 12]      ; N%p
    cmp edx, 0
    je end_media
    neg edx      ; -N%p
    add edx, 4    ; p-N%p

    sub ebx, edx      ; fx*N+i - (p-N%p)
    sub esi, edx      ; i - (p-N%p)

    movups xmm0, [eax + ebx*4] ; [ ds[fx*N+(N-4)] | ds[fx*N+(N-3)] | ds[fx*N+(N-2)] | ds[fx*N+(N-1)] ]
    movups xmm1, xmm0          ; [ ds[fx*N+(N-4)] | ds[fx*N+(N-3)] | ds[fx*N+(N-2)] | ds[fx*N+(N-1)] ]
    movups xmm2, [ecx + esi*4] ; [ labels[N-4] | labels[N-3] | labels[N-2] | labels[N-1] ]

```

In questa prima parte, viene modificato l'indice per considerare gli ultimi 4 elementi di ds e di labels.

```

    movups xmm3, [maskres]    ; residue_mask: 1 1 1 0

    cmp edx, 2                ; p-N%p == 2
    jne res1_media
    insertps xmm3, xmm3, 0x3   ; residue_mask: 1 1 0 0
    jmp calc_res_media

res1_media:
    cmp edx, 3                ; p-N%p == 3
    jne calc_res_media
    insertps xmm3, xmm3, 0x7   ; residue_mask: 1 0 0 0

```

Nel registro xmm3 viene definita la maschera per il residuo a seconda dei possibili di casi di residuo diverso da zero:

Residuo pari a 3	residue_mask			
XMM3	111...1	111...1	111...1	0...0

Residuo pari a 2	residue_mask			
XMM3	111...1	111...1	0...0	0...0

Residuo pari a 1	residue_mask			
XMM3	111...1	0...0	0...0	0...0

Definita la maschera viene applicata sugli elementi di interesse:

```

calc_res_media:
    andps xmm0, xmm3      ; residue_mask on [ ds[fx*N+(N-4)] | ds[fx*N+(N-3)] | ds[fx*N+(N-2)] | ds[fx*N+(N-1)] ]
    andps xmm1, xmm3      ; residue_mask on [ ds[fx*N+(N-4)] | ds[fx*N+(N-3)] | ds[fx*N+(N-2)] | ds[fx*N+(N-1)] ]
    andps xmm2, xmm3      ; residue_mask on [ labels[N-4] | labels[N-3] | labels[N-2] | labels[N-1] ]
    addps xmm5, xmm2      ; counter n1

    xorps xmm4, xmm4      ; 0 0 0 0
    cmpneqps xmm2, xmm4   ; labels_mask based on c=1
    andps xmm2, xmm3      ; residue_mask for labels_mask

    andps xmm0, xmm2      ; ds and labels_mask      (c=1)
    andnps xmm2, xmm1     ; ds and not labels_mask (c=0)

    addps xmm6, xmm0      ; m1 m1 m1 m1
    addps xmm7, xmm2      ; m0 m0 m0 m0

```

La procedura termina con il calcolo e memorizzazione della media e di una parte del rcf della feature. Il valore assoluto viene calcolato utilizzando una maschera che azzera il bit di segno dei 32 bit di interesse.

mask abs				
XMM3	0...0	0...0	0...0	0111...1
ANDPS				
XMM7	-	-	-	mu0-mu1
<hr/>				
XMM7	-	-	-	mu0-mu1

```

    movss [ebx + ecx*4], xmm7 ; mu[fx]=mu0
    add ecx, eax ; fx+d
    movss [ebx + ecx*4], xmm0 ; mu[fx+d]=mu1
    add ecx, eax ; fx+d+d
    movss [ebx + ecx*4], xmm6 ; mu[fx+2*d]=mu

    subss xmm7, xmm0 ; mu0-mu1
    movups xmm3, [maskabs] ; load mask abs
    andps xmm7, xmm3 ; |mu0-mu1|
    divss xmm7, xmm4 ; |mu0-mu1|/N

    sqrtss xmm1, xmm1 ; sqrt(n0)
    sqrtss xmm5, xmm5 ; sqrt(n1)

    subss xmm4, [maskfloat] ; x x x N-1
    sqrtss xmm4, xmm4 ; sqrt(N-1)

    mulss xmm7, xmm1 ; |mu0-mu1|/N * sqrt(n0)
    mulss xmm7, xmm5 ; |mu0-mu1|/N * sqrt(n0) * sqrt(n1)
    mulss xmm7, xmm4 ; |mu0-mu1|/N * sqrt(n0) * sqrt(n1) * sqrt(N-1)

    mov ecx, [ebp + 8] ; fx
    mov ebx, [ebp + 24] ; strcf*
    movss [ebx + ecx*4], xmm7 ; strcf[fx] = |mu0-mu1|/N * sqrt(n0) * sqrt(n1) * sqrt(N-1)

```

## 2.2 RcfP4

La procedura *rcf*, oltre ad ultimare il calcolo di rcf e memorizzare il risultato finale per ogni feature in *strcf*, restituisce anche la feature con *rcf* massima, così da poterla inserire all'interno di *S* come prima feature del risultato.

In *rcf* è stato utilizzato un ciclo innestato: il ciclo esterno itera sulle features considerandone due per volta (*loop unrolling* con fattore pari a 2), mentre il ciclo interno itera sugli elementi considerandone quattro per volta.

```

for_rcf:
    cmp edi, esi ; fi < d-d%
    jge resD_rcf
    mov edx, [ebp+16] ; mu*
    mov ecx, [ebp+20] ; input
    mov ecx, [ecx+24] ; d
    shl ecx, 1 ; 2*d
    add ecx, edi ; 2*d+fi
    movss xmm0, [edx+ecx*4] ; x x x muu (fi)
    shufps xmm0, xmm0, 0 ; muu[] (fi)
    movss xmm4, [edx+ecx*4+4] ; x x x muu (fi+1)
    shufps xmm4, xmm4, 0 ; muu[] (fi+1)

    xorps xmm1, xmm1 ; ximu (fi)
    xorps xmm5, xmm5 ; ximu (fi+1)
    xor eax, eax ; contatore for innestato
    mov ecx, [ebp+20] ; input
    mov edx, [ecx+20] ; N
    mov ebx, [ebp+12] ; N%
    neg ebx ; -N%
    add ebx, edx ; N-N%
    mov ecx, [ecx] ; ds
    mov esi, edi ; copy fi
    inc esi ; fi+1
    imul esi, edx ; (fi+1)*N
    imul edx, edi ; fi*N

forN_rcf:
    cmp eax, ebx ; i < N-N%
    jge resN_rcf

    movups xmm2, [ecx+edx*4] ; ds[fi*N+i...i+4]
    subps xmm2, xmm0 ; x-muu
    mulps xmm2, xmm2 ; (x-muu)*(x-muu)
    addps xmm1, xmm2 ; ximu+= (x-muu)*(x-muu) (fi)

    movups xmm2, [ecx+esi*4] ; ds[(fi+1)*N+i...i+4]
    subps xmm2, xmm4 ; x-muu
    mulps xmm2, xmm2 ; (x-muu)*(x-muu)
    addps xmm5, xmm2 ; ximu+= (x-muu)*(x-muu) (fi+1)

    add eax, 4 ; i+=4
    add edx, 4
    add esi, 4
    jmp forN_rcf

```

La feature con rcf massimo viene conservata in cima allo stack. I confronti fatti per trovare la feature da restituire vengono implementati come segue:

```

comiss xmm2, xmm3           ; strcf[fi] > strcf[fi+1]
jbe cmp_rcf
comiss xmm2, xmm7           ; strcf[fi] > rcfMax
jbe endD_rcf
pop esi
push edi                   ; fmax = fi
movss xmm7, xmm2            ; rcfMax = strcf[fi]
jmp endD_rcf

cmp_rcf:
comiss xmm3, xmm7           ; strcf[fi+1] > rcfMax
jbe endD_rcf
inc edi
pop esi
push edi                   ; fmax = fi+1
dec edi
movss xmm7, xmm3            ; rcfMax = strcf[fi+1]

```

## 2.2 RffP4

La procedura *rff* implementa in Assembly l'algoritmo visto in C, con un ciclo principale ed un ciclo di residuo.

```

for_rff:
    cmp edi, esi                ; i < N-N%p
    jge res_rff

    movups xmm0, [edx+eax*4]      ; ds[fx*N+i...i+4]
    subps xmm0, xmm3              ; x-mux

    movups xmm1, [edx+ebx*4]      ; ds[fy*N+i...i+4]
    subps xmm1, xmm4              ; y-muy

    movups xmm2, xmm0              ; copy x-mux
    mulps xmm2, xmm1              ; (x-mux)*(y-muy)
    addps xmm5, xmm2              ; num num num num

    mulps xmm0, xmm0              ; (x-mux)^2
    mulps xmm1, xmm1              ; (y-muy)^2
    addps xmm6, xmm0              ; den1 den1 den1 den1
    addps xmm7, xmm1              ; den2 den2 den2 den2

    add edi, 4
    add eax, 4                    ; fx*N + i
    add ebx, 4                    ; fy*N + i
    jmp for_rff

```

Il calcolo del residuo usa le stesse logiche discusse nei paragrafi precedenti.

```

res_rff:
    mov esi, [ebp+16]          ; N%p
    cmp esi, 0
    je end_rff
    neg esi                   ; -N%p
    add esi, 4                ; p-N%p
    sub eax, esi               ; fx*N+i - (p-N%p)
    sub ebx, esi               ; fy*N+i - (p-N%p)

    movups xmm0, [edx+eax*4]   ; ds[fx*N+i...i+4]
    subps xmm0, xmm3           ; x-mux
    movups xmm1, [edx+ebx*4]   ; ds[fy*N+i...i+4]
    subps xmm1, xmm4           ; y-muy

    movups xmm3, [maskres]     ; 1 1 1 0
    cmp esi, 2                 ; p-N%p == 2
    jne res1_rff
    insertps xmm3, xmm3, 0x3   ; 1 1 0 0
    jmp calc_res_rff

res1_rff:
    cmp esi, 3                 ; p-N%p == 3
    jne calc_res_rff
    insertps xmm3, xmm3, 0x7   ; 1 0 0 0

calc_res_rff:
    andps xmm0, xmm3          ; x-mux masked
    andps xmm1, xmm3          ; y-muy masked

    movups xmm2, xmm0          ; copy x-mux
    mulps xmm2, xmm1           ; (x-mux)*(y-mux)
    addps xmm5, xmm2           ; num num num num

    mulps xmm0, xmm0           ; (x-mux)^2
    mulps xmm1, xmm1           ; (y-muy)^2
    addps xmm6, xmm0           ; den1 den1 den1 den1
    addps xmm7, xmm1           ; den2 den2 den2 den2

```

Il valore di ritorno viene posto in cima allo stack dei registri x87.

```

end_rff:
    haddps xmm5, xmm5
    haddps xmm5, xmm5          ; num
    haddps xmm6, xmm6
    haddps xmm6, xmm6          ; den1
    haddps xmm7, xmm7
    haddps xmm7, xmm7          ; den2
    mulss xmm6, xmm7           ; den1*den2

    movups xmm3, [maskabs]      ; load mask abs
    andps xmm5, xmm3           ; |num|
    sqrtss xmm6, xmm6          ; sqrt(den1*den2)
    divss xmm5, xmm6            ; rff

    movss [mem_rff], xmm5       ; store rff
    emms                      ; switch to x87
    fld dword [mem_rff]         ; return rff

```

## Confronto Prestazioni Loop-Unroll in C e Code-Vectorization in Assembly

Dal confronto tra la versione che utilizza solo il loop unroll e le versioni che si avvalgono di procedure Assembly emerge come le prestazioni migliorino notevolmente.

Rispetto alla versione loop-unroll in C, risulta molto più impattante l'uso della procedura rff ottimizzata in Assembly rispetto all'uso delle procedure media e rcf. Rff, infatti, è l'istruzione dominante del programma, cioè quella che viene eseguita il maggior numero di volte.

Infine, l'implementazione di tutte e tre le procedure in Assembly permette di non superare i 0,005 secondi.

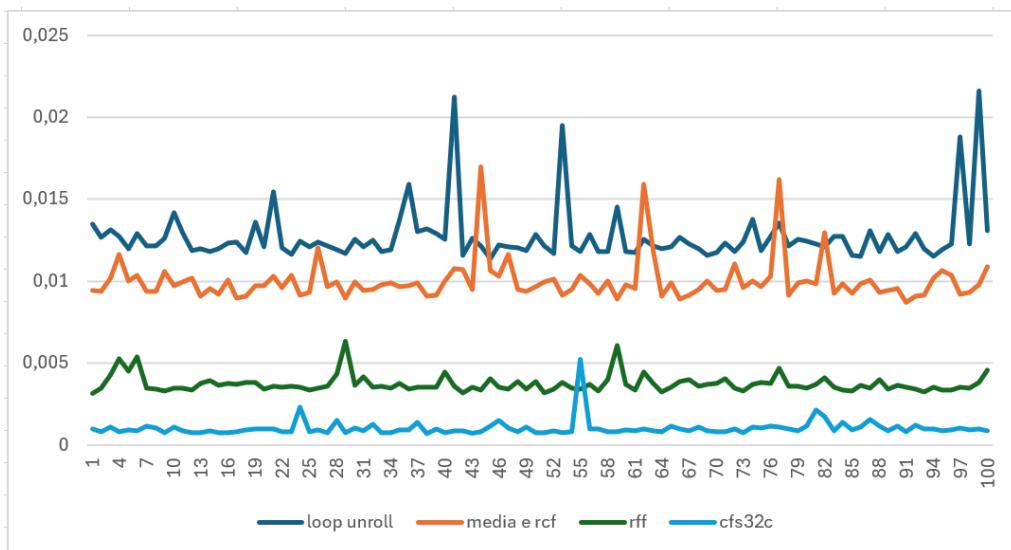


Figura 2 : Confronto prestazioni tra loop unroll in C, media e rcf in Assembly, rff in Assembly e versione completa

## Confronto prestazioni col residuo

Le prestazioni della versione in Assembly sono state testate anche con dataset con residuo diverso da 0. I risultati ottenuti confermano il miglioramento già visto su dataset senza residuo.

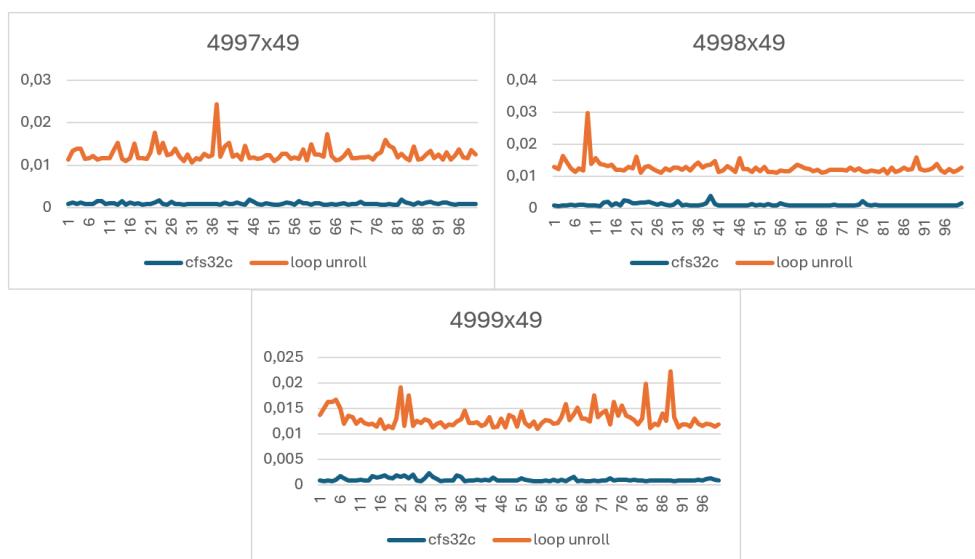


Figura 3: Confronto tra versione completa in Assembly e loop unroll in C con diversi residui

## Versione a 64 bit

Nella versione a 64 bit le procedure sono state definite usando gli stessi meccanismi della versione a 32 bit, ma usando i registri ymm, le istruzioni del repertorio AVX e le corrispondenti convenzioni per il passaggio dei parametri e la restituzione dei valori di ritorno.

Una delle differenze principali riguarda l'operazione di *halfadd* per la riduzione ad un singolo risultato: in questa versione non è possibile eseguire due *hadd* di fila, ma si è reso necessario l'utilizzo di *vextractf128*.

```
end_media:  
    vxorpd ymm0, ymm0  
    vxorpd ymm1, ymm1  
  
    vhaddpd ymm5, ymm5      ; x n1 x n1  
    vextractf128 xmm3, ymm5, 1  
    vaddpd ymm5, ymm3       ; x x x n1
```

Inoltre, l'utilizzo di *vbroadcastsd* ha permesso di semplificare il codice, diminuendo il numero di istruzioni usate.

```
    mov eax, [rcx + 40]          ; d  
    shl rax, 1                 ; 2*d  
    add rax, r10                ; 2*d+fi  
    vbroadcastsd ymm0, [rdx + rax*8]   ; muu muu muu muu (fi)  
    vbroadcastsd ymm4, [rdx + rax*8 + 8] ; muu muu muu muu (fi+1)  
  
    vxorpd ymm1, ymm1          ; ximuu (fi)  
    vxorpd ymm5, ymm5          ; ximuu (fi+1)
```

## Ottimizzazione con OpenMP

La libreria OpenMP consente la creazione di programmi paralleli, fornendo una serie di direttive che permettono al compilatore di avere istruzioni su come parallelizzare il codice. In particolare, calcoliamo le medie, il merit massimo facendo particolare attenzione al suo aggiornamento ed, infine, rffP4 con l'update atomico in modo parallelo.

```
#pragma omp parallel for num_threads(X)
for(int fi=0; fi<input->d; fi++)
    mediaP4(fi, nmodp, mu, input, strcf);

while(cardS<input->k) {
    meritMax=0.0;
    #pragma omp parallel for num_threads(X)
    for(int fx=0; fx<input->d; fx++) {
        if(inS[fx]==0){
            type midrcf = midrcfIn + strcf[fx];
            type midrff = midrffIn + rffC(cardS, fx, mu, input);
            type merit = midrcf / sqrt((cardS+1) + midrff*2);
            #pragma omp critical
            if (meritMax < merit) {
                meritMax = merit;
                fmax = fx;
                midrcfMax = midrcf;
                midrffMax = midrff;
            }
        }
    }

    #pragma omp parallel for num_threads(X)
    for (int y = 0; y < cardS; y++) {
        type tmp = rffP4(fx, input->out[y], nmodp, mu, input);

        #pragma omp atomic update
        ret += tmp;
    }
}

return ret;
```

I seguenti test mostrano come varia il tempo in base al numero di thread, la migliore esecuzione viene fatta sostituendo alla X in ordine 0, 2 e 4.

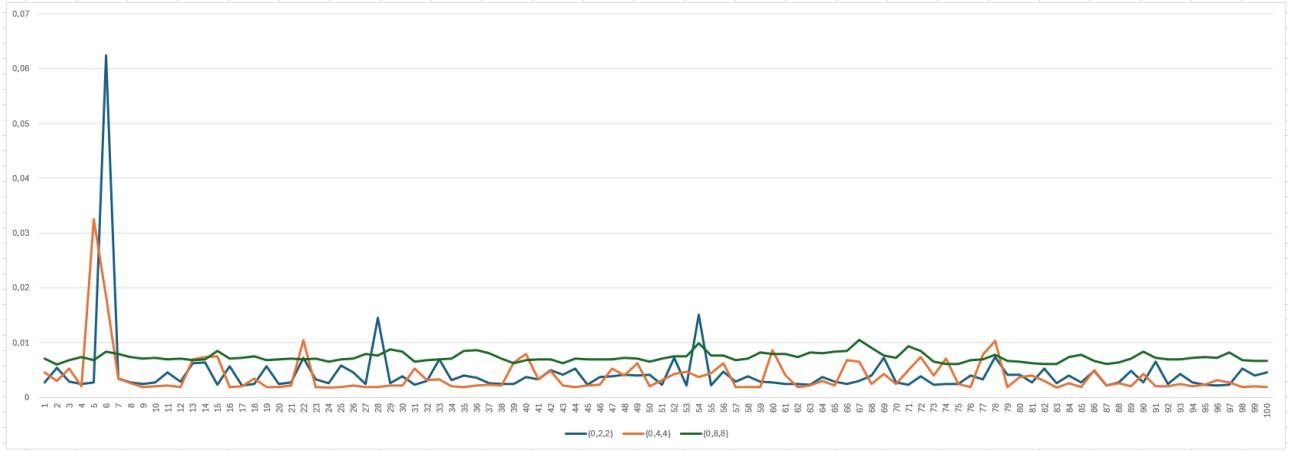


Figura 4: Confronto con 2,2 | 4,4 | 8,8 numero di thread

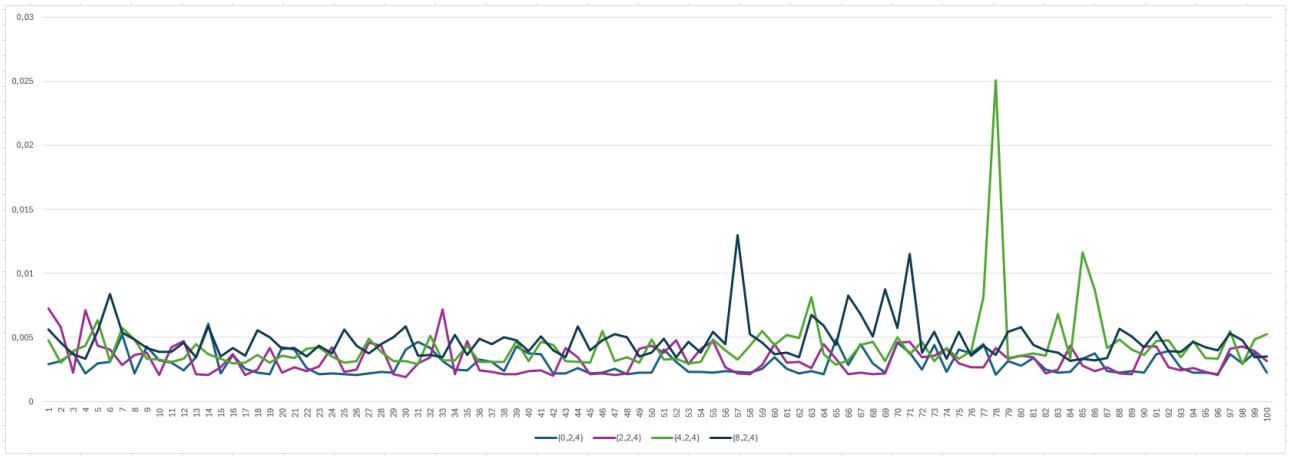


Figura 5: Confronto con 0,2,4| 2,2,4| 4,2,4| 0,8,4 numero di thread

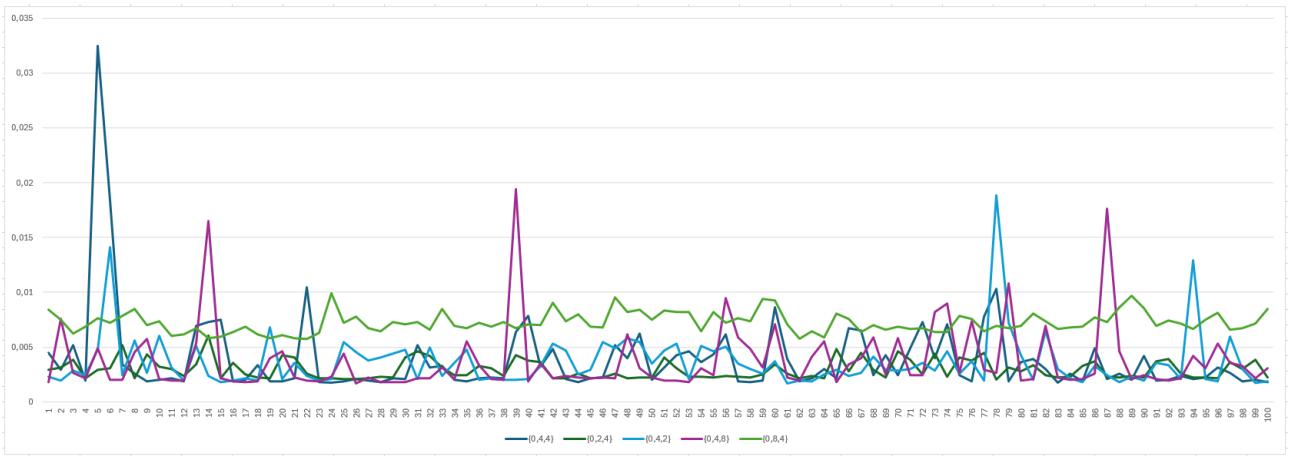


Figura 6: Confronto tra 0,4,4 | 0,2,4 | 0,4,8 | 0,8,4 numero di thread

## Confronto Prestazioni

Il grafico evidenzia che la versione completa in OpenMP presenta picchi occasionali, indicando che in alcuni casi potrebbe essere meno efficiente. Ciò potrebbe essere attribuito al fatto che l'introduzione di direttive OpenMP aumenta l'overhead, in quanto le operazioni di gestione dei thread e della sincronizzazione richiedono risorse aggiuntive, introducendo un costo computazionale aggiuntivo. Questo vale sia per la versione a 32 bit sia per la versione a 64 bit.

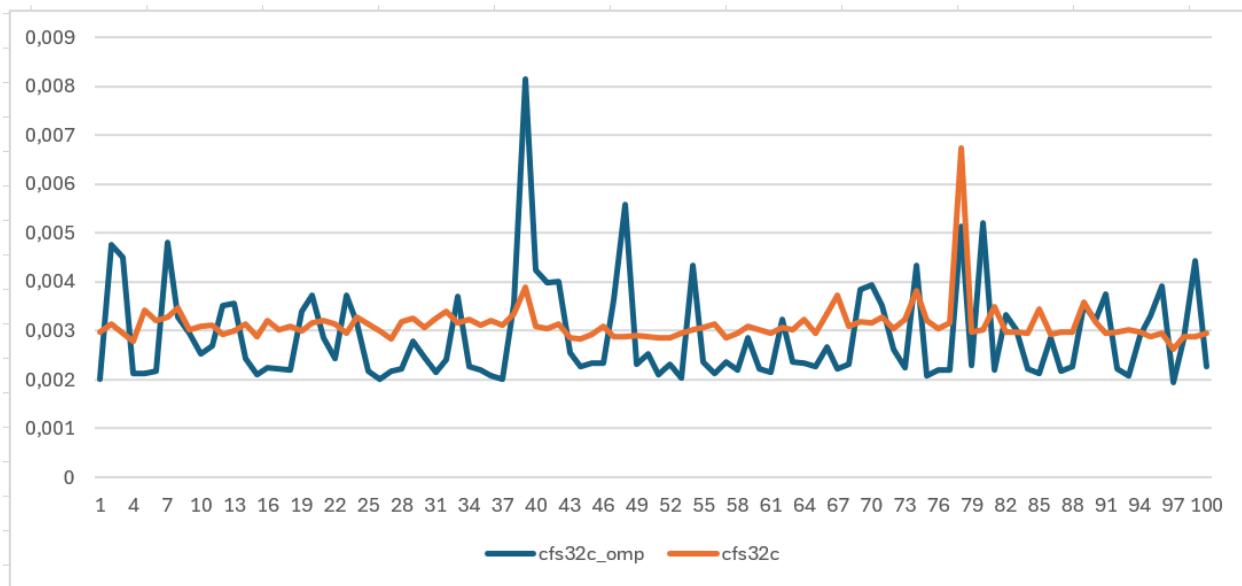


Figura 7: Confronto prestazioni tra versione completa a 32 bit e versione in OMP

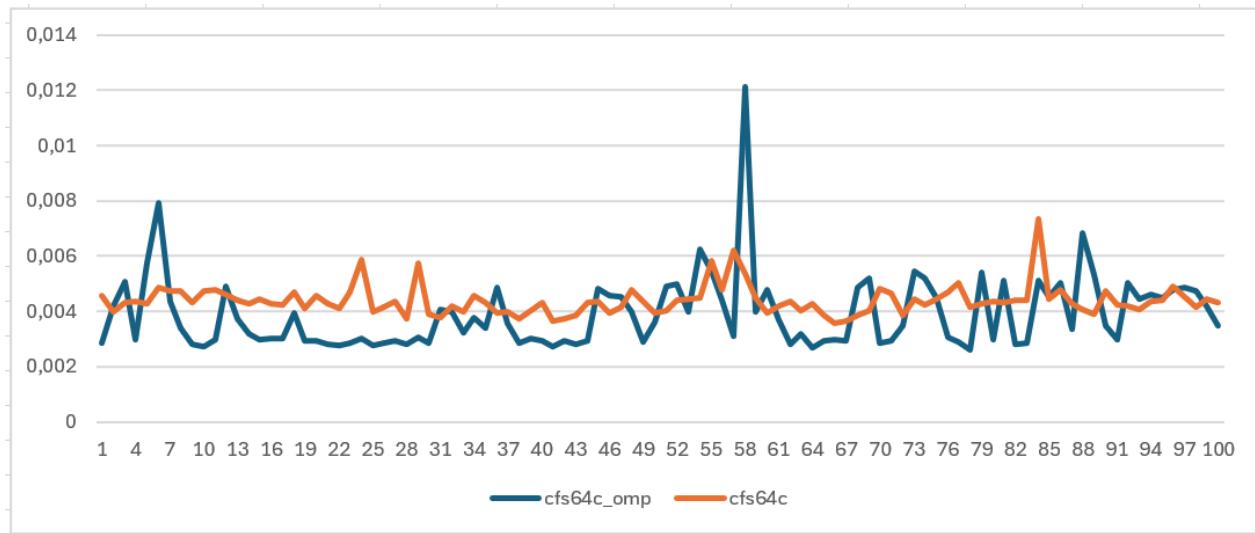


Figura 8: Confronto prestazioni tra versione completa a 64 bit e versione in OMP

## Conclusione

Il nostro progetto si è concentrato sull'implementazione dell'algoritmo Correlation Feature Selection, dedicando particolare attenzione alle prestazioni attraverso l'adozione di tecniche basate sull'utilizzo di operazioni SIMD e l'integrazione con OpenMP.

In dettaglio, abbiamo impiegato istruzioni SIMD come SSE ed AVX, consentendo l'esecuzione simultanea di operazioni su multiple unità di dati. Parallelamente, con l'adozione di OpenMP, una libreria di programmazione parallela, abbiamo sfruttato al massimo le capacità di elaborazione parallela dei moderni processori multi-core.

Attraverso un approccio strategico di loop-unrolling e code-vectorization, abbiamo massimizzato l'efficienza, sfruttando appieno le potenzialità del repertorio dell'architettura x86.

I risultati delle nostre analisi hanno chiaramente evidenziato che l'implementazione di queste ottimizzazioni ha portato a un significativo miglioramento delle prestazioni dell'algoritmo Correlation Feature Selection. La combinazione delle strategie descritte ha garantito un'elevata efficienza nella selezione delle caratteristiche, anche su dataset di dimensioni considerevoli.