Descrizione dell'attività progettuale

Obiettivo del progetto è mettere a punto un'implementazione dell'algoritmo di *Corner Detection di Harris* e dell'algoritmo di *Filtraggio Spaziale* in linguaggio C e di migliorarne le prestazioni utilizzando le tecniche di ottimizzazione basate sull'organizzazione dell'hardware. In particolare, l'algoritmo riceve l'immagine f e deve poter funzionare in modalità:

- **Corner detection**, nel qual caso sarà opzionalmente possibile specificare i parametri σ, θ e t, e verranno restituite in uscita la matrice di risposta R e l'elenco dei corner di f;
- **Filtering**, nel qual caso dovrà ricevere in ingresso una seconda immagine w rappresentante il filtro e verrà restituita in uscita l'immagine g = f ⋆ w.

L'ambiente sw/hw di riferimento è costituito dal linguaggio di programmazione C (gcc), dal linguaggio assembly x86-32+SSE e dalla sua estensione x86-32+AVX (nasm) e dal sistema operativo Linux (ubuntu).

Di seguito si riportano le linee guida per il corretto svolgimento del progetto:

- Inannzitutto, codificare l'algoritmo interamente in linguaggio C, possibilmente in maniera modulare (ovvero come sequenza di chiamate a funzioni);
- Sostituire una o piu` (idealmente tutte le) funzioni scritte in linguaggio ad alto livello con corrispondenti funzioni scritte in linguaggo assembly.
- Sono richieste due soluzioni software, la prima per l'architettura x86-32+SSE e la seconda per l'architettura x86-64+AVX. Per la codifica dei numeri in virgola mobile utilizzare la precisione singola (32 bit per numero).

Progettazione

La fase d'implementazione è stata ovviamente preceduta da quella di progettazione. In questa fase si è analizzato l'algoritmo di Corner Detection (che fa uso del filtraggio spaziale) per determinarne i colli di bottiglia e dunque focalizzare l'attenzione sulle operazioni più onerose. In particolare, come suggerito dalla traccia, si è notato che utilizzando la caratteristica del filtro separabile è possibile ridurre drasticamente i tempi di calcolo su alcune operazioni del Corner Detection. Inoltre si è cercato di stabilire delle linee guide da utilizzare in fase d'implementazione per scrivere il codice in maniera modulare. La suddivisione in moduli ha semplificato, come si vedrà di seguito, l'analisi delle singole funzioni facilitandone lo studio dettagliato dei tempi di esecuzione.

Implementazione

C

La prima versione del software è stata realizzata interamente in C senza alcuna ottimizzazione. Questo ha facilitato l'analisi delle prestazioni per ogni miglioria apportata al codice. Si è deciso di implementare il codice in maniera modulare così da avere un riscontro diretto sui tempi di calcolo delle diverse funzioni (creazioni matrice di smoothing, filtraggio spaziale, derivate parziali, ecc.). Questo è stato fondamentale nelle fasi successive poiché si è partiti ad analizzare e ottimizzare proprio i metodi più onerosi.

Grazie alla modularità e dunque allo studio dei tempi di ogni singolo metodo, si è notato subito che la funzione richiedente più tempo computazionale è proprio quella che esegue il filtraggio spaziale. Come descritto dalla traccia, il software deve funzionare in due modalità: **corner detection** e **filtraggio spaziale**. L'algoritmo di corner detection fa uso del filtraggio ma con la particolarità di applicare un determinato filtro (filtro Gaussiano). Si è deciso dunque di lavorare su metodi separati e specializzati.

Filtraggio spaziale

Per quanto riguarda la modalità **Filtraggio Spaziale** si è riusciti a eliminare alcune operazioni di controllo suddividendo l'immagine da filtrare in più parti e richiamando un metodo specifico per ognuno di essi.

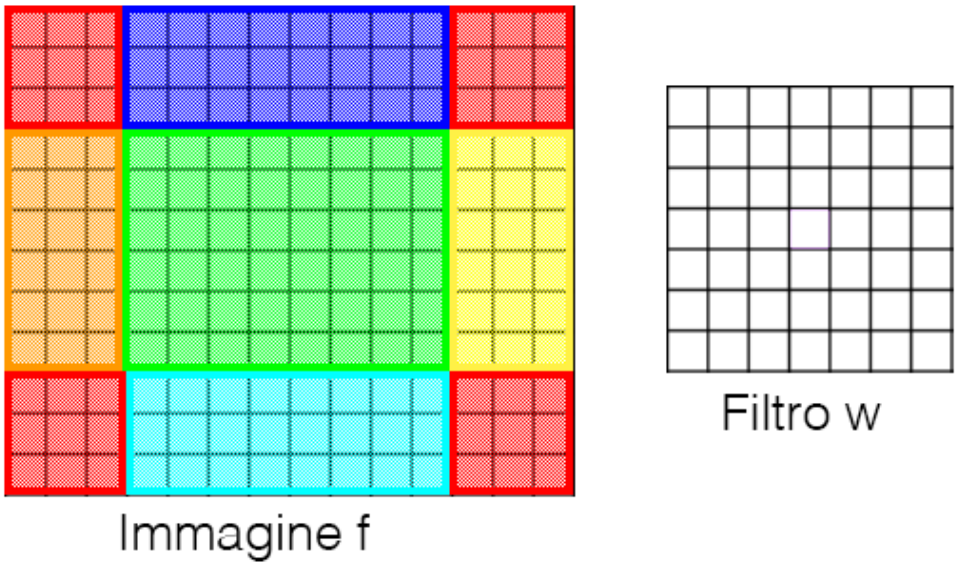

Figura 1: Divisione matrice per l'applicazione del filtro spaziale. Ogni area è gestita da un metodo specializzato.

Così facendo si sono evitati molti controlli nei casi in cui il filtro, una volta sovrapposto all'immagine, sforasse. Se ad esempio si prende la sezione nord (riquadro blu) si nota che il filtro applicato a una cella all'interno di quest'area può sforare solo verso l'alto, dunque si fa un solo controllo sulla riga. Stesso discorso vale per le altre sezioni sui bordi. Sui vertici invece (riquadri rossi) vanno eseguiti più controlli poiché può sforare sia sulle righe sia sulle colonne. La dimensione dei "bordi" della matrice dipende dal filtro da applicare ed è equivalente a **(n-1)/2**,

dove **n** è la dimensione del filtro. È facile intuire che più l'immagine è grande e più si notano i benefici portati da quest'ottimizzazione.

Filtraggio con filtro gaussiano

Il filtraggio spaziale appena visto è stato utilizzato e testato anche nella modalità di **Corner Detection**, cioè nell'applicazione dei filtraggi con filtri gaussiani. Si è in seguito deciso di utilizzare una caratteristica dei filtri gaussiani: la *separabilità*. Questa caratteristica ha permesso la realizzazione di un'ulteriore ottimizzazione nell'algoritmo di corner detection con un notevole risparmio di tempo. Il primo passo è la realizzazione di un metodo per la creazione del vettore di smoothing. Sapendo che il vettore è sempre simmetrico si è evitato di eseguire il doppio dei calcoli. È stato necessario importare la libreria *math.h* per l'uso di π e della funzione *sqrt*.

```
float* smooth2(float theta,int n) {
          int np=n>>1;
          float* ris=(float*)malloc(n*sizeof(float));
          float smooth1=1/((sqrt(2*M_PI)*theta));
          float smooth2=-2*theta*theta;
          int nc=n>>1;
          int nm1=n-1;
          float t, sum=0;
          int inp;
          for(int i=0; i<nc; i++) {
                    inp=i-np;
                    t=smooth1*exp((inp*inp)/smooth2);
                    ris[i]=t;
                    ris[nm1-i]=t;
                    sum+=2*t;
          }
          ris[nc]=smooth1*exp(((nc-np)*(nc-np))/smooth2);
          sum+=ris[nc];
          for(int i=0; i<nc; i++) {
                    ris[i]/=sum;
                    ris[nm1-i]=ris[i];
          }
          ris[nc]/=sum;
          return ris;
}
```

Dopo aver creato il vettore di smoothing (opportunamente normalizzato), si è passati all'implementazione di una funzione specializzata per il filtraggio con filtro gaussiano. Infatti, grazie al fatto che il filtro gaussiano gode della proprietà di separabilità, anziché applicare l'intera matrice di Gauss, è possibile invece applicare due volte il vettore generato precedentemente. È necessario dunque applicare una volta il filtro orizzontalmente e una seconda volta verticalmente. Anche in questo caso si è resi conto della possibilità di evitare molti controlli nel caso in cui il filtro sforasse. L'immagine seguente rappresenta, come visto in precedenza con il filtro spaziale generale, la suddivisione della matrice in più sezioni.

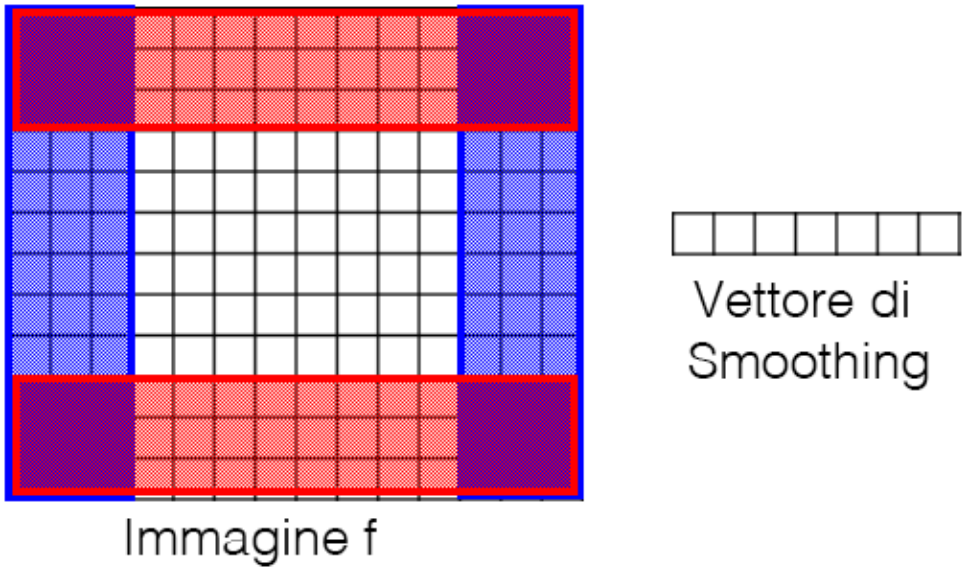

Figura 2: Divisione matrice per l'applicazione del filtro di smoothing.

Il filtro orizzontale sforerà solo nelle celle della matrice all'interno delle sezioni blu (viceversa per il filtro verticale). Evitare i controlli sulle dimensioni e posizioni nelle aree dove i filtri non sforano, permette di ridurre i tempi di calcolo.

Derivate parziali

Un altro metodo ottimizzato è quello che calcola le derivate parziali. Normalmente si potrebbe usare la funzione di filtro spaziale generale, cioè applicando all'immagine il filtro *dx* e *dy*, ma a una seconda analisi ci si accorge che molti calcoli sono inutili. Infatti, si a che fare solo con moltiplicazioni per 0, 1 e -1. Vedendo moltiplicazioni di questo tipo viene naturale pensare di trasformarle in addizioni e sottrazioni. È stato dunque implementato un metodo specializzato.

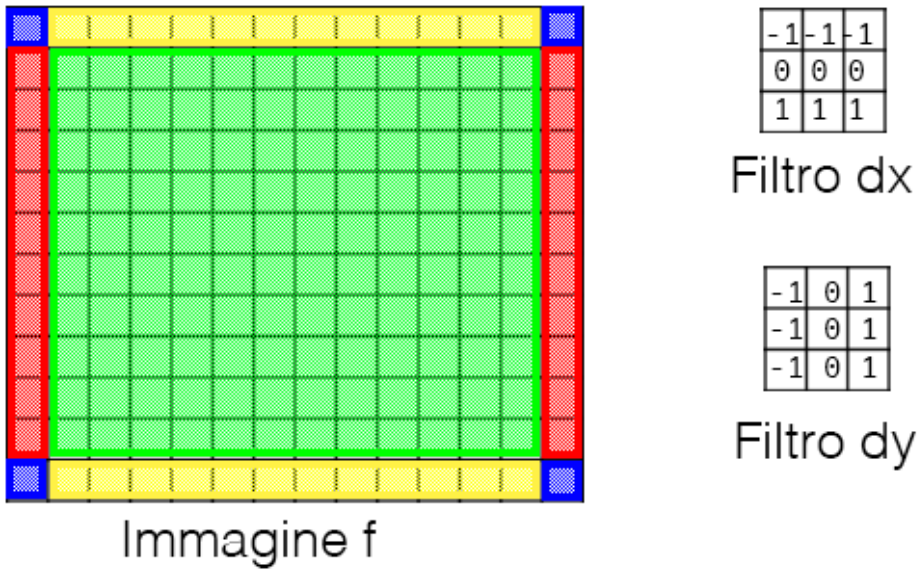

Figura 3: Divisione matrice per l'applicazione del filtro derivata. I bordi sono gestiti separatamente dal resto della matrice.

Si è suddivisa la matrice in più aree per gestire le celle in cui i filtri sforassero. Le sezioni rosse corrispondono alla prima e ultima colonna della matrice, escludendo le celle della prima e ultima

riga. Di seguito, utilizzando per motivi di semplicità la notazione `f[i][j]` al posto di quella effettivamente usata nel codice `f[i*m+j]`, si mostra il funzionamento della derivata parziale (filtraggio con filtro *dx*) nelle celle all'interno delle sezioni rosse:

```
for(i=1;i<n-1;i++){
        fx[i][0]=-f[i-1][0]-f[i-1][0]-f[i-1][1]+f[i+1][0]+f[i+1][0]+f[i+1][1];
        fx[i][m-1]=-f[i-1][m-2]-f[i-1][m-1]-f[i-1][m-1]+f[i+1][m-2]+f[i+1][m-1]+f[i+1][m-1];
        fy[i][0]=-f[i-1][0]+f[i-1][1]-f[i][0]+f[i][1]-f[i+1][0]+f[i+1][1];
        fy[i][m-1]=-f[i-1][m-2]+f[i-1][m-1]-f[i][m-2]+f[i][m-1]-f[i+1][m-2]+f[i+1][m-1];
}
```

Si nota subito la potenza di quest'ottimizzazione poiché si è passati da una somma di 9 prodotti a 6 somme algebriche. Stesso discorso vale per le sezioni gialle (prima e ultima riga). I quattro vertici della matrice (sezioni blu) sono stati calcolati singolarmente. Di seguito è mostrato il calcolo della cella [0,0] della derivata *dx*:

```
fx[0][0]=-f[0][0]-f[0][0]-f[0][1]+f[1][0]+f[1][0]+f[1][1];
```

I valori interni (sezione verde) sono stati calcolati come segue:

```
for(i=1; i<n-1; i++) {
        for(j=1; j<m-1; j++) {
                fx[i][j]=-f[i-1][j-1]-f[i-1][j]-f[i-1][j+1]+f[i+1][j-1]+f[i+1][j]+f[i+1][j+1];
                fy[i][j]=-f[i-1][j-1]+f[i-1][j+1]-f[i][j-1]+f[i][j+1]-f[i+1][j-1]+f[i+1][j+1];
        }
}
```

SSE

Dopo aver implementato in C una versione funzionante e aver introdotto alcune ottimizzazioni, si è cercato di ottenere ulteriori migliorie nelle prestazioni utilizzando il linguaggio a basso livello **NASM** e l'estensione per l'architettura x86 **SSE** (SIMD Streaming Extensions).

Prodotto

La funziona che più si presta a un'ottimizzazione a basso livello è la funzione che **moltiplica due matrici elemento per elemento**. Nel realizzare questo metodo sono stati adottati due meccanismi di fondamentale importanza per il miglioramento delle prestazioni: il ***loop vectorization*** e il ***code vectorization***. Il loop vectorization è una tecnica che lavora sulle iterazioni eseguite dai cicli e sarà spiegato più avanti. Il code vectorization sfrutta le istruzioni vettoriali messe a disposizione da SSE per effettuare operazioni su più valori con un ridotto numero di istruzioni. In particolare, utilizzando i registri xmm a 128 bit e le operazioni vettoriali movups (unaligned packed single-precision move) e mulps (packed single-precision multiplication), è stato possibile prelevare dalla memoria e moltiplicare 4 float con sole due istruzioni.

```
%include "init.nasm"
section .data
        f equ       8                   ; matrice f
        g equ       12                  ; matrice g
        n equ       16                  ; rows
        m equ       20                  ; cols
        ris         equ       24        ; matrice ris
        dim         equ       4         ; dimensione ogni locazione
```

```
        pdim        equ       16
        pdim2       equ       32
        pdim3       equ       48
section .bss
section .text
global prodotto
prodotto: go
        mov eax,[ebp+n]
        imul eax,[ebp+m]     ; n*m
        mov ebx,16
        mov edx,0
        div ebx              ; eax=quoziente , edx=resto
        mov ecx,eax          ; ecx=quoziente
        mov esi, [ebp+f]     ; indirizzo di f
        mov edi, [ebp+g]     ; indirizzo di g
        mov eax, [ebp+ris]
        cmp ecx, 1
        jl .cicloR
.cicloQ:
        movups xmm1, [esi]            ; prelevo f[i:i+3]
        movups xmm2, [edi]            ; prelevo g[i:i+3]
        mulps xmm1,xmm2
        movups [eax],xmm1
        movups xmm3, [esi+pdim]       ; prelevo f[i+4:i+7]
        movups xmm4, [edi+pdim]       ; prelevo g[i+4:i+7]
        mulps xmm3,xmm4
        movups [eax+pdim],xmm3
        movups xmm5, [esi+pdim2]      ; prelevo f[i+8:i+11]
        movups xmm6, [edi+pdim2]      ; prelevo g[i+8:i+11]
        mulps xmm5,xmm6
        movups [eax+pdim2],xmm5
        movups xmm0, [esi+pdim3]      ; prelevo f[i+12:i+15]
        movups xmm7, [edi+pdim3]      ; prelevo g[i+12:i+15]
        mulps xmm0,xmm7
        movups [eax+pdim3],xmm0
        add esi, 64
        add edi, 64
        add eax, 64
        dec ecx
        jnz .cicloQ
        cmp edx, 0
        jz .fine
.cicloR:
        movss xmm1, [esi]    ; prelevo f[i]
        movss xmm2, [edi]    ; prelevo g[i]
        mulss xmm1,xmm2
        movss [eax],xmm1
        add esi, 4
        add edi, 4
        add eax, 4
        dec edx
        jnz .cicloR
        .fine:
end
```

La funzione è composta di due cicli detti *quoziente* (cicloQ) e *resto* (cicloR). Il numero d'iterazioni è calcolato facendo la divisione tra il numero di celle di una delle matrice (le matrici hanno dimensione uguale) e 16. Il risultato della divisione rappresenta il numero d'iterazioni del ciclo quoziente mentre il resto indica quelle del ciclo resto. Il ciclo quoziente utilizza operazioni vettoriali e moltiplica, a ogni iterazione, 16 valori prelevati dalle due matrici di input. Il ciclo resto fa uso d'istruzioni scalari (movss e mulss) e gestisce, se ce ne sono, i valori restanti.

Sogliatura

Un altro metodo convertito da C a NASM è la funzione che esegue la **sogliatura** della matrice.

```
%include "init.nasm"
section .data
        f           equ       8         ; matrice f
        n           equ       12        ; rows
        m           equ       16        ; cols
        s           equ       20        ; soglia
        dim         equ       4         ; dimensione ogni locazione
section .bss
section .text
global sogliatura
sogliatura: go
        mov ecx,[ebp+n];
        imul ecx,[ebp+m];
        mov edx,[ebp+s];
        mov esi, [ebp+f]     ; indirizzo di f
        mov ebx, 0
.for:
        mov eax, [esi+ebx]   ; prelevo f[i]
        sub eax, edx;
        jge .nonazzero
        mov [esi+ebx], dword 0
        .nonazzero:
        add ebx, dim
        dec ecx
        jnz .for
end
```

Per la natura stessa delle funzione non è stato possibile applicare le ottimizzazioni implementate sulla funzione prodotto, poiché è necessario controllare elemento per elemento. Preso un elemento della matrice lo si confronta con il valore di soglia e, dopo aver effettuato un confronto, se risulta minore lo si azzera.

AVX

Una volta completata l'implementazione del codice per l'architettura x86-32 si è passati a realizzare quella per processori x86-64, sfruttando i nuovi registri messi a disposizione da questa architettura e le ulteriori istruzioni introdotte con l'avvento della tecnologia Intel® AVX (Advanced Vector Extension). I registri *xmm* (ora chiamati *ymm*) dell'architettura precedente sono stati raddoppiati passando da 128 bit a 256 bit e ne sono stati introdotti altri 8, per un totale di 16.

Prodotto

È stato naturale pensare di ottimizzare la funzione **prodotto** sfruttando la dimensione, ora raddoppiata, dei nuovi registri ymm e le istruzioni vettoriali dedicate.

```
%include "init.nasm"
%include "sseutils64.nasm"
section .data
f       equ     8       ; matrice f ->rdi
g       equ     16      ; matrice g ->rsi
n       equ     24      ; rows->rdx
m       equ     32      ; cols->rcx
ris     equ     40      ; matrice ris -> r8
dim     equ     8       ; dimensione ogni locazione
p       equ     4       ; parallel
pdim    equ     32
```

```
pdim2    equ      64
pdim3    equ      96
section .bss
section .text
global prodotto
prodotto: go
       mov rax,rdx
       imul rax,rcx         ; n*m
       mov rbx,32
       xor rdx,rdx
       idiv rbx             ; rax=quoziente , rdx=resto
       mov rcx,rax          ; rcx=quoziente
       cmp rcx, 1
       jl .cicloR
.cicloQ:
       vmovups ymm1, [rdi]           ; prelevo f[i:i+7]
       vmovups ymm2, [rsi]           ; prelevo g[i:i+7]
       vmulps ymm1,ymm2
       vmovups [r8],ymm1

       vmovups ymm3, [rdi+pdim]      ; prelevo f[i+8:i+15]
       vmovups ymm4, [rsi+pdim]      ; prelevo g[i+8:i+15]
       vmulps ymm3,ymm4
       vmovups [r8+pdim],ymm3

       vmovups ymm5, [rdi+pdim2]     ; prelevo f[i+16:i+23]
       vmovups ymm6, [rsi+pdim2]     ; prelevo g[i+16:i+23]
       vmulps ymm5,ymm6
       vmovups [r8+pdim2],ymm5

       vmovups ymm0, [rdi+pdim3]     ; prelevo f[i+24:i+31]
       vmovups ymm7, [rsi+pdim3]     ; prelevo g[i+24:i+31]
       vmulps ymm0,ymm7
       vmovups [r8+pdim3],ymm0

       add rsi, 128
       add rdi, 128
       add r8, 128
       sub rcx,1
       cmp rcx,0
       jg .cicloQ
       cmp rdx, 0
       jle .fine
.cicloR:
       vmovss xmm1, [rdi]   ; prelevo f[i]
       vmovss xmm2, [rsi]   ; prelevo g[i]
       vmulss xmm1,xmm2
       vmovss [r8],xmm1
       add rsi, 4
       add rdi, 4
       add r8, 4
       sub rdx,1
       jg .cicloR
.fine:
end
```

Con registri a 128 bit e le istruzioni SSE si riesce a moltiplicare 4 float con una singola operazione, mentre con registri a 256 bit e le istruzioni AVX si riesce a raddoppiare questo valore gestendo 8 float per volta. Questo comporta un calo dei tempi di esecuzione della funzione **prodotto**, ma in generale non porta notevoli migliorie all'intero codice poiché è una funzione utilizzata poco frequentemente ed inoltre non è tra le funzioni più costose in termini di tempi di esecuzione.

Valutazione prestazioni

Ogni nuova versione del codice consisteva di due fasi: implementazione e analisi. Nella prima è stato scritto il codice avendo cura di mantenere la modularità tra le funzioni e testando singolarmente ogni metodo. Nella seconda invece si è assicurato il corretto funzionamento dell'intero software mettendo insieme le varie funzioni. Inoltre, sempre in questa fase, si sono salvati i tempi di esecuzione così da avere un riscontro tra le varie versioni. Il grafico seguente è stato realizzando eseguendo il codice in modalità Corner Detection sul file "*ananas.img*" utilizzando una macchina dotata di 8GB di RAM, processore Intel® Core™ i5-2430 M (3M Cache, 2.00 GHz) e sistema operativo Ubuntu Linux 17.04.

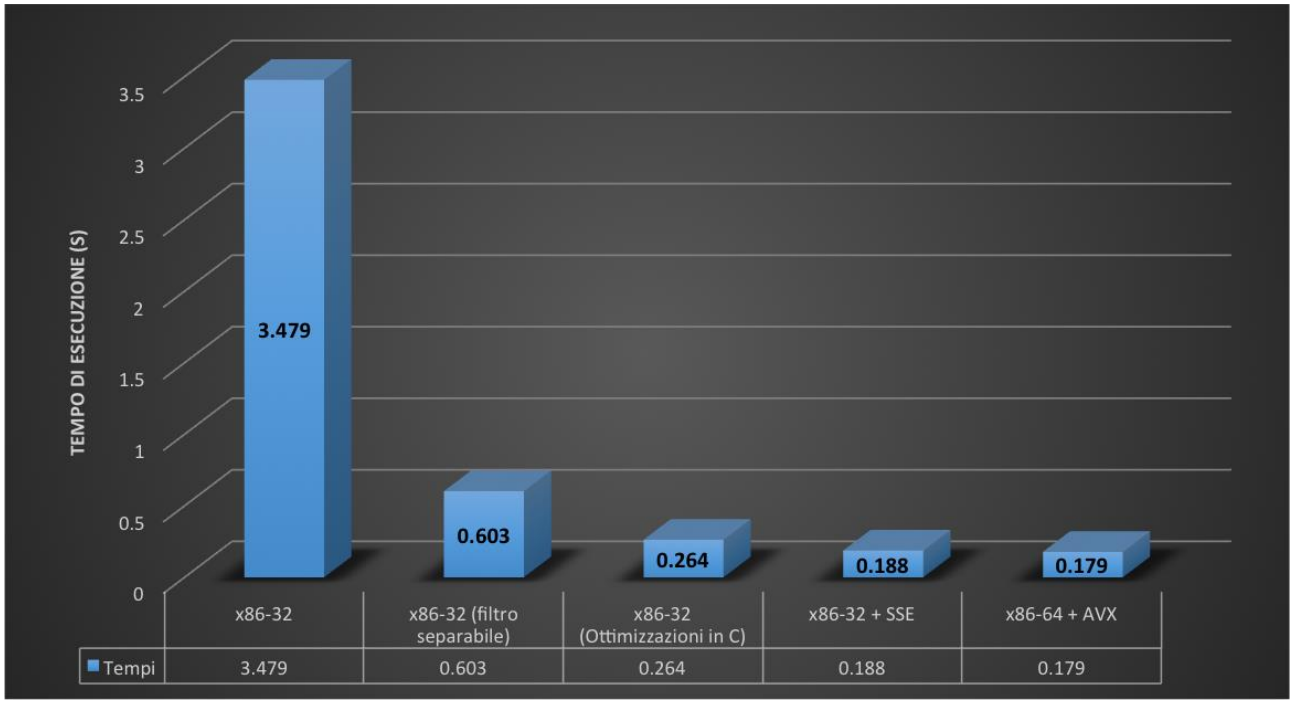

Figura 4: Rappresentazione delle prestazioni delle differenti versioni del codice.

1° versione (x86-32): Codice C

Interamente scritta in C ed esente da ottimizzazioni, ha riscontrato un tempo di esecuzione di diversi secondi. Essendo la prima versione si è cercato di scrivere un codice funzionante da poter analizzare per scovare i colli di bottiglia. Questo ha portato ad individuare la funzione più onerosa in termini di calcolo: il filtraggio spaziale.

2° versione (x86-32): Ottimizzazione filtro separabile in C

Avendo identificato la funzione più costosa si è cercato di migliorarla applicando delle ottimizzazioni o addirittura riscrivendola da zero. Dato che il filtro da applicare gode della proprietà di separabilità (filtro gaussiano) si è deciso di sfruttare questa caratteristica per ridurre notevolmente il numero di operazioni eseguite. Ne ha conseguito una drastica riduzione del tempo di esecuzione.

3° versione (x86-32): Accorgimenti nei cicli for in C

Prima di passare all'utilizzo della programmazione a basso livello si è cercato di ottimizzare ulteriormente il codice usando piccoli accorgimenti nei cicli e nell'inizializzazione delle variabili. Segue un esempio:

```
for(int i=0; i<n; i++) {
  for(int j=0; j<m; j++) {
           f[i*m+j]=...;
  }
}
```

```
int i,j,im,imj;
im=-m;
for(i=0; i<n; i++) {
  im+=m;
  imj=im-1;
  for(j=0; j<m; j++) {
           imj++;
           f[imj]=...;
  }
}
```

Questa porzione di codice racchiude numerosi accorgimenti. Innanzitutto le variabili sono inizializzate una sola volta, prima dei cicli di *for*. Il calcolo della posizione della riga non avviene più come prodotto `(i*m)` poiché è stato trasformato in una somma `(im+=m)` e avviene solo a ogni ciclo esterno. Queste ottimizzazioni sono state applicate a ogni ciclo *for* e hanno portato una riduzione non trascurabile al tempo complessivo d'esecuzione.

4° versione (x86-32): Metodi in NASM + SSE

L'uso dell'estensione SSE ha permesso di ottimizzare la funzione *prodotto* avendo reso disponibile l'uso di tecniche come il **code vectorization** ed il **loop vectorization**.

5° versione (x86-64): Metodi in NASM + AVX

In quest'ultima fase si è fatto uso dell'estensione AVX per migliorare ulteriormente le prestazioni della funzione dedicata al calcolo del prodotto degli elementi delle matrici.

Considerazioni Finali

Come suggerito dalla traccia, si è iniziato a scrivere il codice in C per poi cercare di sfruttare le estensioni SSE e AVX per migliorarne le prestazioni. L'unica funzione che è stata ottimizzata sfruttando queste estensioni è stata quella che più si prestava alla vettorizzazione delle operazioni, cioè il prodotto degli elementi di due matrici. Tutte le altre funzioni (derivate parziali, smoothing, filtraggio spaziale, sogliatura, soppressione, ecc.) sono state lasciate in C poiché una loro implementazione in NASM non ha restituito benefici in termini prestazionali, bensì in alcuni casi si è notato un degrado dei tempi prestazionali. Molte delle funzioni non avrebbero ottenuto alcun risultato a basso livello poiché già ottimizzate in C e una loro conversione in NASM sarebbe stata abbastanza complessa essendo il numero dei registri limitato, ma anche penalizzante nelle prestazioni essendo un adattamento del codice.