

Progetto GPU Computing - HOG

Marco Cacciatori 980909, Francesco Stella 979050

Marzo 2022

1 Introduzione

Il seguente progetto è volto a presentare i possibili speed-up ottenibili nel calcolo dell'Histogram of Oriented Gradients (HOG) in immagini e video tramite l'impiego di Graphics Processing Units (GPUs).

Nel seguito viene presentato l'approccio seguito per il calcolo dell'HOG, motivando le scelte effettuate ed eseguendo un confronto fra i tempi di esecuzione del codice su CPU e su GPU, presentando inoltre un'analisi dettagliata di alcune metriche per quest'ultima architettura.

2 Panoramica del progetto

L'obiettivo del progetto è quello di realizzare un algoritmo per il calcolo dell'HOG e di parallelizzare tale algoritmo per utilizzare le risorse della GPU. L'algoritmo è composto da una serie di passi, sintetizzabili tramite la pipeline mostrata in **figura 1**. Ad ogni passo della pipeline vi è la possibilità di salvare un'immagine di output per visionare il risultato dei singoli step. Inoltre, poichè è necessario un trattamento specifico per i video dati in input all'algoritmo, discuteremo nella **sezione 3** la procedura seguita per gestirli e per poter applicare l'algoritmo nel modo descritto nelle successive sezioni.

Il primo step della pipeline consiste nella conversione dell'immagine data in input in scala di grigi, immediatamente seguito dall'applicazione dell'algoritmo di gamma correction per amplificare la differenza fra i valori bassi e quelli alti della scala di grigi. Discuteremo tali dettagli nella **sezione 4**.

Nella **sezione 5** parleremo del calcolo dei gradienti orizzontale e verticale, che viene eseguito sull'immagine restituita dalla procedura di gamma correction.

La magnitudine e la direzione dei gradienti vengono calcolate partendo dai due gradienti e verranno trattate nella **sezione 6**.

Infine, nella **sezione 7** parleremo dell'Histogram of Oriented Gradients, che viene ottenuto partendo da magnitudine e direzione dei gradienti precedentemente calcolati.

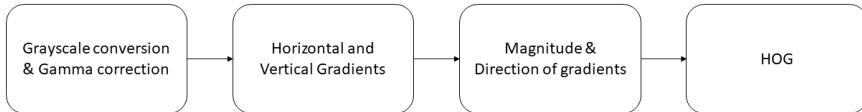


Figure 1: Passi seguiti dall'algoritmo per il calcolo dell'HOG.

3 Estrazione dei frames dai video

Nel caso in cui l'input dato all'algoritmo è un video, risulta necessaria l'estrazione dei singoli frames appartenenti al video stesso per poter applicare l'algoritmo del calcolo dell'HOG. Per realizzare tale operazione è stata utilizzata una libreria esterna, *FFMPEG*, i cui output vengono salvati come immagini indipendenti, poi fornite all'algoritmo in modo sequenziale e continuativo. Il problema dell'applicazione dell'algoritmo HOG sui video viene quindi ridotto all'applicazione dell'algoritmo stesso su una sequenza più o meno lunga di immagini, che rappresentano i frames del video.

La gestione di input video è la medesima sia nel caso della CPU che nel caso della GPU, dunque il guadagno in termini di tempo di elaborazione è dato dalla differenza fra i tempi cumulati di elaborazione delle singole immagini da parte della CPU e della GPU.

4 Grayscale Conversion & Gamma Correction

4.1 Descrizione algoritmi e applicazione su CPU

Per convertire un'immagine RGB in un'immagine in scala di grigi è stato utilizzato il metodo ponderato. Poichè i tre diversi colori (rosso, verde, blu) hanno diverse lunghezze d'onda, dobbiamo pesare la media in funzione del loro contributo. Dato che il colore rosso ha lunghezza d'onda maggiore rispetto agli altri due colori, utilizzando il metodo ponderato [4] (o metodo di luminosità) il suo contributo viene diminuito, mentre quello del colore verde viene aumentato. Quindi, per trasformare un pixel RGB nel suo omologo in scala di grigi è stata



(a) Immagine originale RGB. (b) Immagine in scala di grigi

Figure 2: Conversione da RGB a grayscale

utilizzata la **formula 1**.

Secondo questa formula, il rosso ha contribuito per circa il 30 percento, il verde per il 59 percento (che è anche il contributo maggiore di tutti e tre i colori) e il blu per l'11 percento.

$$((0,299 * R) + (0,587 * G) + (0,114 * B)) \quad (1)$$

Applicando questa formula all'immagine originale, otteniamo il risultato di **figura 2**.

L'impiego dell'algoritmo di **Gamma Correction** è invece più articolato e necessita di qualche attenzione aggiuntiva. Fondamentalmente, questo si compone di sei step principali [3], come indicato di seguito:

1. Sia $f(x, y)$ il valore di intensità dell'immagine di input in corrispondenza del pixel di coordinate (x, y) ;
2. Calcolare l'istogramma cumulativo C per l'immagine di input tramite le equazioni **2** e **3**;
3. Trovare i valori di minimo, massimo e la mediana nel range $C_5 - C_{95}$;
4. Calcolare il valore gamma g tramite la formula 4;
5. Normalizzare g tramite la formula 5;

6. Applicare la gamma correction sull'immagine di input f usando il valore normalizzato g per produrre l'immagine a contrasto migliorato G , tramite l'equazione 6.

In corrispondenza del punto 2. occorre creare l'istogramma dei livelli di intensità dell'immagine e per realizzarlo è stato utilizzato un range del singolo bin pari a $L=4$, ovvero vi sono 4 valori di intensità per singolo bin:

$$his(r_k) = n_k \quad (2)$$

Dove r_k è un livello di intensità ed n_k è il numero di pixel con intensità r_k . Ottenuto tale istogramma, si procede con il calcolo dell'istogramma cumulato:

$$C(H_j) = \sum_{j=1}^k (his(r_j)) \quad (3)$$

In cui k è un livello di intensità fissato, $his(r_j)$ è il numero di pixel con livello di intensità pari a j e $C(H_j)$ è il valore per il livello j -esimo nell'istogramma cumulato.

Una volta ottenuto l'istogramma cumulato C è possibile calcolare il minimo, il massimo e la mediana considerando il range che va dal 5% al 95% dei pixel nell'istogramma C .

Il calcolo del valore di g è dato dalla formula seguente:

$$g = \log\left(\frac{\text{mediana} - \text{minimo}}{\text{massimo} - \text{mediana}}\right) \quad (4)$$

Successivamente, tale valore viene normalizzato:

$$gNorm = \begin{cases} 0.8 & \text{se } g < 0.8 \\ 1.2 & \text{se } g > 1.2 \\ g & \text{altrimenti} \end{cases} \quad (5)$$

Infine, l'immagine a contrasto migliorato è ottenuta applicando il valore gamma normalizzato con la seguente formula:

$$G(x, y) = f(x, y)^{\frac{1}{g}} \quad (6)$$

L'applicazione di tale algoritmo su CPU non richiede particolari accorgimenti. Nella prossima sezione esploriamo l'implementazione su GPU.

4.2 Applicazione su GPU

La conversione in scala di grigi in versione GPU fa esclusivamente uso della global memory. Tale scelta è dettata dal fatto che l'applicazione della **formula 1** avviene per ogni pixel e non implica letture o scritture multiple sullo stesso dato, dunque l'utilizzo della global memory è stato valutato come la miglior opzione.

Per l'applicazione della Gamma Correction sono necessari alcuni accorgimenti, in quanto bisogna innanzitutto creare l'istogramma cumulato e poi calcolare ed applicare la gamma correction tramite il valore normalizzato g . Per calcolare g abbiamo bisogno del numero totale di pixel ricadenti in ciascun bin dell'istogramma, che a sua volta contiene $256/L = 64$ bins, in quanto L è stato fissato a 4. Poichè vi sono molteplici blocchi che lavorano simultaneamente sull'immagine in input, dobbiamo trovare un modo efficiente per risalire alla massima intensità a livello globale (considerando l'intera immagine). È stata utilizzata la **shared memory** per memorizzare l'istogramma e la massima intensità a livello di blocco, in modo da velocizzare le operazioni di scrittura e lettura grazie alla ridotta latenza. Delle **operazioni atomiche** sono necessarie per aggiornare il conteggio dei pixel per ogni bin a livello di blocco, mentre l'intensità massima dei pixel elaborati dai threads del blocco è ottenuta usando le funzioni di **warp shuffle**, per scambiare in modo efficiente una variabile fra i threads di un warp.

In particolare, avendo fissato una dimensione del blocco pari a 64 threads, sono stati considerati due **indici di warp**, ad ognuno dei quali sono associati 32 **indici di lane**. I due threads con *lane ID* pari a 0 sono stati utilizzati per recuperare localmente le intensità massime relative al proprio warp, per poi effettuare un'operazione atomica di massimo fra i due per recuperare l'intensità massima a livello di blocco. In questo modo si è evitato di eseguire 64 operazioni atomiche, che risulterebbero eccessivamente dispendiose. La figura 3 mostra come vengono utilizzati gli indici di lane e di thread, insieme alla shared memory, per il recupero della massima intensità a livello di blocco.

Una volta ottenuto l'istogramma dei livelli di intensità, si calcola su CPU l'istogramma cumulato ed il valore di g *normalizzato*. Infine, la gamma correction viene applicata utilizzando la GPU e la formula 5.

L'implementazione della Gamma Correction su GPU è stata testata con l'impiego del solo *null-stream* e, successivamente, con due e quattro streams. Un confronto dei tempi medi di esecuzione è riportato in figura 14 e mostra una significativa differenza fra l'utilizzo del solo null-stream e di 2 o 4 streams.

```

if(threadIdx.x == 0)
    s_intensity = 0;

unsigned int intensity = (unsigned int)img_gray[i];
__syncthreads(); // s_num must be set entirely to 0

atomicAdd((unsigned int *)&s_num[(intensity/L)], 1);

for(int srcLane=1; srcLane<32; srcLane++) {
    int srcLaneVal = __shfl(intensity, srcLane);
    if(laneIdx == 0 && srcLaneVal > intensity)           // only threads 0 and 32
        intensity = srcLaneVal;
}

__syncthreads();

atomicAdd((unsigned int *)&num[threadIdx.x], s_num[threadIdx.x]);

if(laneIdx == 0) {
    atomicMax((unsigned int *)&s_intensity, intensity);
}
__syncthreads();

// Now s_intensity has the maximum intensity related to the current block

```

Figure 3: Parte di codice del kernel per il calcolo della massima intensità dell’immagine a livello di blocco.

5 Calcolo dei gradienti

5.1 Calcolo dei gradienti su CPU

Il calcolo dei gradienti orizzontale e verticale viene eseguito sull’output dei precedenti steps (sezione 4). L’operatore scelto per calcolare le derivate orizzontale e verticale è l’operatore di *Sobel*, definito nel modo seguente:

$$S_X = \begin{pmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{pmatrix}$$

$$S_Y = \begin{pmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

Un aspetto a cui occorre prestare attenzione riguarda il calcolo dei gradienti lungo il bordo dell’immagine. Infatti, essendo l’operatore di Sobel una matrice 3x3, nel caso in cui questo sia centrato su un pixel del bordo, i valori esterni non risultano definiti. Per il calcolo su CPU, l’operatore di Sobel viene applicato su tutti i pixel non appartenenti ai bordi dell’immagine, in modo tale da avere l’operatore completamente all’interno dell’immagine. La figura 5 mostra la situazione limite, in cui l’operatore è al bordo dell’immagine. In figura 4 sono mostrati i risultati dell’applicazione dell’operatore di Sobel ad un’immagine tramite l’implementazione per CPU.



(a) Gradiente orizzontale dell'immagine.



(b) Gradiente verticale dell'immagine.

Figure 4: Risultato del calcolo del gradiente dell'immagine ottenuto tramite applicazione dell'operatore di Sobel su CPU.

5.2 Calcolo dei gradienti su GPU

Il calcolo dei gradienti su GPU prevede l'utilizzo della *Shared Memory* (SMEM), che fornisce una latenza sensibilmente inferiore rispetto alla Global Memory, nonché l'utilizzo della *Constant Memory* per la memorizzazione dell'operatore di Sobel. Il caricamento dei dati da Global a Shared memory avviene seguendo il *row-major order*, con l'intento di evitare possibili conflitti all'interno dei banchi della SMEM. Il codice mostrato in figura 6 è un esempio di caricamento dei dati in SMEM.

La scelta di utilizzare la constant memory, invece, deriva dalla necessità di eseguire letture ripetute sugli elementi dell'operatore stesso. La constant memory è una memoria che risiede nella DRAM (come la global memory) ma ha una cache dedicata on-chip, read-only da parte del device. La constant memory dovrebbe fornire una latenza sensibilmente inferiore rispetto alla Global memory quando tutti thread di un warp accedono in modo uniforme ai suoi dati. Nel nostro caso è stato testato l'impiego sia della Global memory che della Constant memory per la memorizzazione dell'operatore di Sobel, senza significative differenze. Tuttavia, è stata mantenuta l'implementazione con Constant memory. Per il calcolo dei gradienti orizzontale e verticale è stato utilizzato un unico kernel. Ciò è stato possibile grazie alla proprietà di *separabilità* dell'operatore

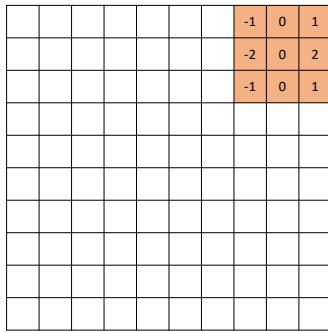


Figure 5: L'operatore di Sobel S_X è applicato tramite convoluzione all'immagine. Poichè i valori esterni al bordo dell'immagine non sono definiti, l'operatore è in una situazione limite e non può essere spostato ulteriormente a destra. Il prossimo step lo vede spostato in basso di una cella e allineato al bordo sinistro dell'immagine.

```
// Top side of the block
if ((threadIdx.y < radius) ) {

    // top left corner of the block
    if (threadIdx.x < radius && (x-radius) >= 0 && (y-radius) >= 0)
        img_shared[threadIdx.y][threadIdx.x] = input_image[(y-radius) * width + x - radius];

    // top right corner of the block
    if (threadIdx.x >= block_m_radius && (x+radius) < width && (y-radius) >= 0)
        img_shared[threadIdx.y][threadIdx.x + 2*radius] = input_image[(y-radius) * width + x + radius];

    // top side of the block
    if ((y-radius) >= 0)
        img_shared[threadIdx.y][threadIdx.x + radius] = input_image[(y-radius) * width + x];
}

}
```

Figure 6: Esempio di caricamento di parte dell'immagine da Global a Shared memory.

```

int sum_x = 0;
int sum_y = 0;
for (int i = 0; i < MASK_SIZE; i++) {
    for (int j = 0; j < MASK_SIZE; j++) {
        sum_x += img_shared[threadIdx.y + i][threadIdx.x + j] * sobelX[i*MASK_SIZE + j];
        sum_y += img_shared[threadIdx.y + i][threadIdx.x + j] * sobelY[i*MASK_SIZE + j];
    }
}

```

Figure 7: Codice per l'applicazione dell'operatore di Sobel tramite GPU su parte dell'immagine originale.

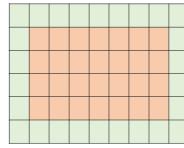


Figure 8: Rappresentazione di una tile in SMEM. La parte arancione ha le stesse dimensioni del blocco di threads, mentre la parte verde rappresenta l'alone aggiuntivo contenente i dati dei blocchi adiacenti. In tal modo è possibile applicare l'operatore di Sobel direttamente in SMEM.

di Sobel, ovvero la possibilità di applicarlo indipendentemente in entrambe le direzioni. Il codice in figura 7 mostra l'applicazione dell'operatore di Sobel su GPU ad una specifica regione dell'immagine originale.

La shared memory è stata allocata *staticamente* con una dimensione rettangolare. Sono state testate diverse dimensioni per la SMEM e per i blocchi di threads: i risultati migliori relativi alla gld_efficiency e alla gls_efficiency sono stati ottenuti con dimensioni per i blocchi pari a $block.x=32$ e $block.y=16$. Di conseguenza, la SMEM ha dimensioni pari a 18 e 34, a causa dell'utilizzo delle *tiles*. Queste sono necessarie in quanto la shared memory è accessibile e condivisa solamente da threads di un dato blocco b , dunque abbiamo bisogno di un alone aggiuntivo per memorizzare i valori dei pixel esterni alla porzione di immagine gestita da b in SMEM. La figura 8 riassume tale concetto. Poiché l'operatore di Sobel (matrice 3x3) ha un raggio pari a 1, le tiles avranno le dimensioni del blocco incrementate di 2.



(a) Gradiente orizzontale dell'immagine.



(b) Gradiente verticale dell'immagine.

Figure 9: Risultato del calcolo del gradiente di un'immagine tramite applicazione dell'operatore di Sobel su GPU.

6 Magnitudine e direzione dei gradienti

Successivamente al calcolo dei gradienti orizzontale e verticale, si procede con il calcolo della magnitudine e della direzione del gradiente. La magnitudine è la radice dei quadrati del gradiente orizzontale e verticale, come in equazione 7:

$$g = \sqrt{g_x^2 + g_y^2} \quad (7)$$

La direzione del gradiente rappresenta la direzione verso cui si ha la maggior variazione di intensità del gradiente ed è data dall'equazione 8:

$$\theta = \arctan \frac{g_y}{g_x} \quad (8)$$

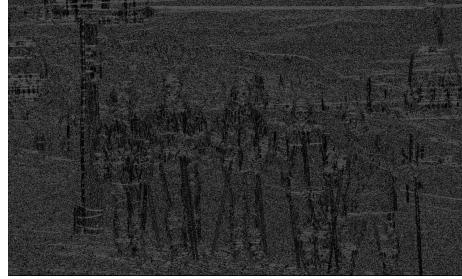
La dimensione del blocco è stata fissata a 32 thread, in quanto ha fornito i migliori valori per la gld_efficiency e la gst_efficiency.

7 Histogram of Oriented gradients

L'HOG è un descrittore di features largamente usato in Computer Vision che consente di svolgere tasks come l'*object detection*. L'input dell'algoritmo consiste nella magnitudine e nella direzione del gradiente descritte in sezione 6.



(a) Magnitudine del gradiente.



(b) Direzione del gradiente.

Figure 10: Risultati del calcolo della magnitudine e della direzione del gradiente tramite l'implementazione per CPU.

L'idea è quella di dividere l'immagine in blocchi di dimensione 8x8, 16x16 oppure 32x32 e, avendo come input le due matrici dello step precedente, ciascuna di esse viene divisa in blocchi. Successivamente, si considerano al più 9 bins per ogni blocco [1], ognuno dei quali rappresenta un range pari a $\theta=20^\circ$, per un totale di 180° che rappresenta il limite massimo dell'orientamento dell'edge, assumendo che tale orientamento sia privo di segno. Per ogni blocco, si considera poi il contributo che ogni pixel apporta ai 9 bins, nel modo seguente:

- Consideriamo due pixel corrispondenti, di coordinate (i,j) , nelle due matrici date in input. Indichiamo con $m_{i,j}$ e con $d_{i,j}$ rispettivamente la magnitudine e la direzione del gradiente associate al pixel (i,j) .
- Il contributo del pixel (i,j) viene spartito fra due bins, che chiamiamo l ed u , calcolati nel modo seguente:

$$l = \frac{d_{i,j}}{\theta} \quad (9)$$

$$u = l + 1 \quad (10)$$

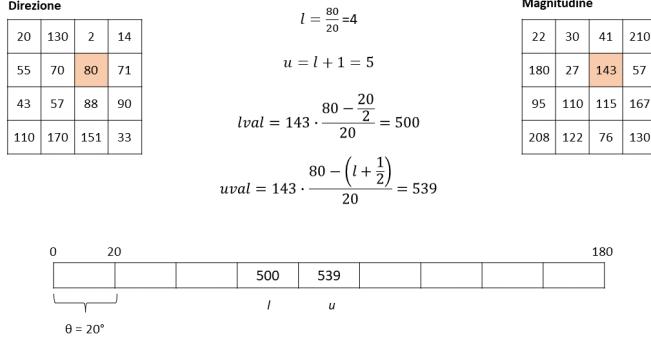


Figure 11: Esempio di applicazione dell’algoritmo HOG. Il calcolo dei bin (l ed u) e dei contributi corrispondenti è mostrato per le celle colorate.

Se il valore di u è maggiore di 9, allora viene impostato a 0.

- I valori associati ai bins l ed u , che chiamiamo rispettivamente $lval$ ed $uval$, vengono calcolati nel modo seguente:

$$lval = m_{i,j} \cdot \frac{d_{i,j} - \frac{\theta}{2}}{\theta} \quad (11)$$

$$uval = m_{i,j} \cdot \frac{d_{i,j} - \left(l + \frac{1}{2}\right)}{\theta} \quad (12)$$

Al termine avremo i descrittori delle features sotto forma degli istogrammi calcolati per ogni blocco. La figura 11 mostra uno step di esempio di applicazione dell’algoritmo HOG.

7.1 HOG su CPU

L’implementazione dell’algoritmo per il calcolo dell’HOG su CPU segue direttamente dalla descrizione appena fornita, senza particolari accorgimenti.

7.2 HOG su GPU

Per quanto riguarda l’implementazione dell’algoritmo per il calcolo dell’HOG su GPU, vi sono alcuni aspetti da considerare.

Per la scelta della dimensione del blocco, la *load/store efficiency* è stata valutata utilizzando dimensioni del blocco rispettivamente pari a 8, 16 e 32. Una dimensione del blocco pari a 32 ha portato alla massima efficienza media.

S.O.	CPU	GPU	RAM
Ubuntu 20.04	intel core i7-7700 3.60GHz	Nvidia GTX 1060 3GB	16GB

Table 1: Specifiche hardware del computer utilizzato per l'esecuzione degli esperimenti.

	CPU	GPU (1s)	GPU (2s)	GPU (4s)
immagini	0.107	0.004	0.003	0.003
speed-up immagini	0%	96.3%	97.2%	97.2%
video	59.05	2.04	1.76	1.59
speed-up video	0%	96.54%	97.02%	97.31%

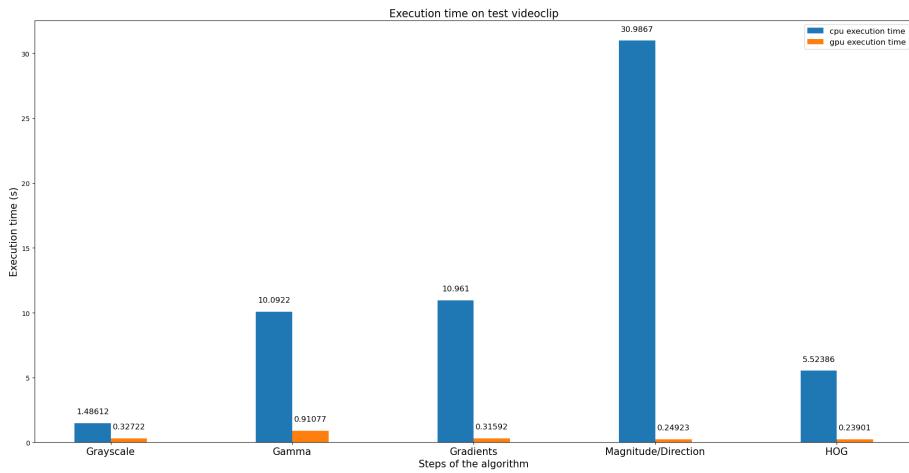
Table 2: Tempi di esecuzione (in secondi) e speed-up ottenuti per ogni ambito di applicazione dell'algoritmo (immagini o video). Per le immagini, i tempi di esecuzione e gli speed-up sono da intendersi come valori medi. N.B.: 1s, 2s o 4s (accanto alla voce GPU) indicano il numero di streams impiegati.

Nell'implementazione del kernel vi è, tuttavia, la presenza di due *atomicAdd()*, necessarie per l'aggiornamento dei bins.

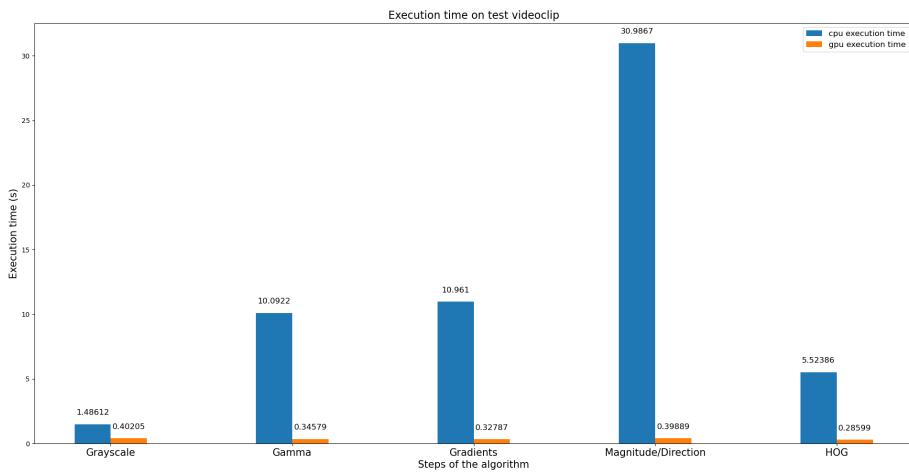
8 Esperimenti e risultati

Le implementazioni per CPU e GPU sono state confrontate tramite l'utilizzo di immagini e video. In particolare, l'algoritmo è stato eseguito iterativamente su un sottoinsieme di 12 immagini dell'INRIA Person Dataset [2], per un totale di 5 ripetizioni. Successivamente, la media aritmetica dei tempi di esecuzione di ogni step dell'algoritmo è stata calcolata e confrontata fra i due device. Per l'implementazione GPU, i tempi di esecuzione medi sono stati calcolati con un numero diverso di streams, come rappresentato in figura 14. Da notare che il calcolo dei gradienti (secondo step in figura 1) e dell'HOG (quarto step in figura 1) avviene utilizzando solamente il null-stream (o default stream) per semplificare l'implementazione.

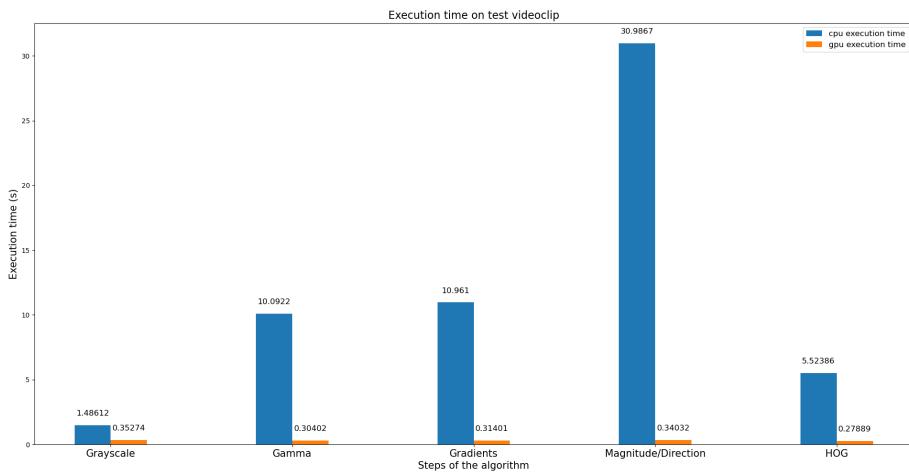
Inoltre, un breve filmato è stato utilizzato per un ulteriore confronto fra le implementazioni CPU e GPU. Sono stati estratti i primi 480 frames di un video caratterizzato da 24 frames/secondo (ovvero i primi 20 secondi) e sono stati calcolati i tempi di esecuzione cumulati per ogni step dell'algoritmo. In questo modo è stato possibile calcolare la differenza totale nei tempi di esecuzione su CPU e GPU ed ottenere lo **speed-up** per tale video, come mostrato in tabella 2. La figura 12 mostra tali tempi di esecuzione. Infine, in tabella 1 è riportata la configurazione hardware usata per gli esperimenti, mentre in figura 13 sono riportate alcune metriche per la GPU.



(a) Confronto CPU - GPU con 1 stream.



(b) Confronto CPU - GPU con 2 streams.



(c) Confronto CPU - GPU con 4 streams.

Figure 12: Confronto dei tempi di esecuzione delle implementazioni per CPU e per GPU su un video di prova.

```

==22363== Profiling application: ./main -i images/crop001007.png
==22363== Profiling result:
==22363== Metric result:
Invocations Metric Name Metric Description Min Max Avg
Device "NVIDIA GeForce GTX 1060 3GB (0)"
Kernel: convolutions gpu(unsigned char*, unsigned char*, unsigned char*, int, int)
    1 achieved occupancy Achieved Occupancy 0.974711 0.974711 0.974711
    1 gld efficiency Global Memory Load Efficiency 15.53% 15.53% 15.53%
    1 gst efficiency Global Memory Store Efficiency 57.00% 57.00% 57.00%
Kernel: create_hist gpu(unsigned int, unsigned char*, unsigned int*, unsigned long)
    1 achieved occupancy Achieved Occupancy 0.973256 0.973256 0.973256
    1 gld efficiency Global Memory Load Efficiency 100.00% 100.00% 100.00%
    1 gst efficiency Global Memory Store Efficiency 0.00% 0.00% 0.00%
Kernel: hog gpu(float*, unsigned char*, unsigned char*, int, int)
    1 achieved occupancy Achieved Occupancy 0.920496 0.920496 0.920496
    1 gld efficiency Global Memory Load Efficiency 57.00% 57.00% 57.00%
    1 gst efficiency Global Memory Store Efficiency 0.00% 0.00% 0.00%
Kernel: apply_gamma gpu(unsigned char*, double, double, unsigned long)
    1 achieved occupancy Achieved Occupancy 0.732809 0.732809 0.732809
    1 gld efficiency Global Memory Load Efficiency 100.00% 100.00% 100.00%
    1 gst efficiency Global Memory Store Efficiency 100.00% 100.00% 100.00%
Kernel: grayscale_gpu(unsigned char*, unsigned char*, unsigned long)
    1 achieved occupancy Achieved Occupancy 0.866352 0.866352 0.866352
    1 gld efficiency Global Memory Load Efficiency 33.33% 33.33% 33.33%
    1 gst efficiency Global Memory Store Efficiency 100.00% 100.00% 100.00%
Kernel: mag_dir_gpu(unsigned char*, unsigned char*, unsigned char*, unsigned char*, unsigned long)
    1 achieved occupancy Achieved Occupancy 0.936528 0.936528 0.936528
    1 gld efficiency Global Memory Load Efficiency 100.00% 100.00% 100.00%
    1 gst efficiency Global Memory Store Efficiency 100.00% 100.00% 100.00%

```

(a) Metriche acquisite con GPU su un'immagine di prova.

```

==23379== Profiling application: ./main -v /home/francesco/Scaricati/video_test.mp4
==23379== Profiling result:
==23379== Metric result:
Invocations Metric Name Metric Description Min Max Avg
Device "NVIDIA GeForce GTX 1060 3GB (0)"
Kernel: convolutions gpu(unsigned char*, unsigned char*, unsigned char*, int, int)
    9 achieved occupancy Achieved Occupancy 0.972343 0.976596 0.974553
    9 gld efficiency Global Memory Load Efficiency 16.14% 16.14% 16.14%
    9 gst efficiency Global Memory Store Efficiency 100.00% 100.00% 100.00%
Kernel: create_hist gpu(unsigned int, unsigned char*, unsigned int*, unsigned long)
    9 achieved occupancy Achieved Occupancy 0.972730 0.973694 0.973173
    9 gld efficiency Global Memory Load Efficiency 100.00% 100.00% 100.00%
    9 gst efficiency Global Memory Store Efficiency 0.00% 0.00% 0.00%
Kernel: hog_gpu(float*, unsigned char*, unsigned char*, int, int)
    9 achieved occupancy Achieved Occupancy 0.913235 0.916022 0.914603
    9 gld efficiency Global Memory Load Efficiency 100.00% 100.00% 100.00%
    9 gst efficiency Global Memory Store Efficiency 0.00% 0.00% 0.00%
Kernel: apply_gamma_gpu(unsigned char*, double, double, unsigned long)
    9 achieved occupancy Achieved Occupancy 0.729831 0.731783 0.731103
    9 gld efficiency Global Memory Load Efficiency 100.00% 100.00% 100.00%
    9 gst efficiency Global Memory Store Efficiency 100.00% 100.00% 100.00%
Kernel: grayscale_gpu(unsigned char*, unsigned char*, unsigned long)
    9 achieved occupancy Achieved Occupancy 0.857292 0.877195 0.869022
    9 gld efficiency Global Memory Load Efficiency 33.33% 33.33% 33.33%
    9 gst efficiency Global Memory Store Efficiency 100.00% 100.00% 100.00%
Kernel: mag_dir_gpu(unsigned char*, unsigned char*, unsigned char*, unsigned char*, unsigned long)
    9 achieved occupancy Achieved Occupancy 0.924202 0.930324 0.926962
    9 gld efficiency Global Memory Load Efficiency 100.00% 100.00% 100.00%
    9 gst efficiency Global Memory Store Efficiency 100.00% 100.00% 100.00%

```

(b) Metriche acquisite con GPU su video di prova (10 frames).

Figure 13: Load/store efficiency su immagine e video di prova. E' stato utilizzato solamente il null-stream.

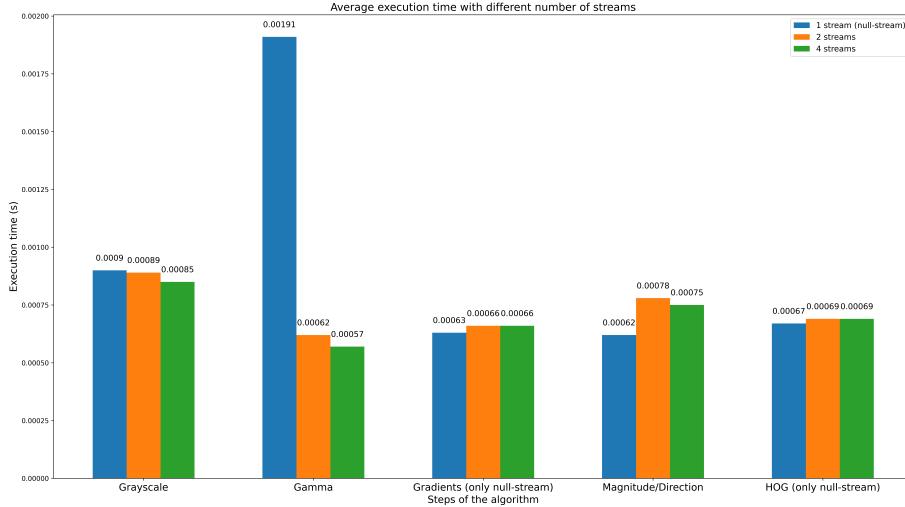


Figure 14: Confronto dei tempi medi di esecuzione su GPU con l'utilizzo di un diverso numero di streams. Per il calcolo dei gradienti e dell'HOG viene utilizzato esclusivamente lo stream di default.

9 Conclusioni

Il presente lavoro ha mostrato la possibilità di ottenere uno speed-up, in termini di tempo di elaborazione, durante l'applicazione dell'algoritmo per il calcolo dell'HOG tramite l'uso di GPU. Alcune statistiche riguardanti l'utilizzo della GPU sono state presentate, insieme agli eventuali accorgimenti adottati per migliorare l'utilizzo della GPU.

E' opportuno precisare che sono possibili ulteriori miglioramenti, come un miglioramento della load efficiency durante il calcolo dei gradienti e l'impiego della libreria NVENC/NVDEC per l'estrazione dei frames dai video, senza la necessità di passare dalla memoria secondaria che rappresenta un collo di bottiglia. Tuttavia, nonostante tale libreria non sia stata impiegata, è stato possibile ottenere uno speed-up significativo.

References

- [1] N. Dalal and B. Triggs. “Histograms of oriented gradients for human detection”. In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’05)*. Vol. 1. 2005, 886–893 vol. 1. DOI: [10.1109/CVPR.2005.177](https://doi.org/10.1109/CVPR.2005.177).
- [2] INRIA. *INRIA Person Dataset*. URL: <http://pascal.inrialpes.fr/data/human/>.
- [3] Somasundaram Karuppanagounder and Kalavathi Palanisamy. “Medical Image Contrast Enhancement based on Gamma Correction”. In: *International Journal of Knowledge Management and e-Learning* 3 (Feb. 2011), pp. 15–18.
- [4] Mathworks. *RGB to Grayscale conversion*. URL: <https://it.mathworks.com/help/matlab/ref/im2gray.html>.