Università degli Studi di Napoli "Parthenope"
Facoltà di Scienze e Tecnologie
Corso di laurea in Informatica Applicata

PROGETTO DI CALCOLO PARALLELO E DISTRIBUITO 2

Applicazione per il calcolo della Trasformata Discreta di Haar di immagini in CUDA

Davide Sito
Matricola 0120000129

Anno Accademico 2013-2014

Indice

1 Introduzione

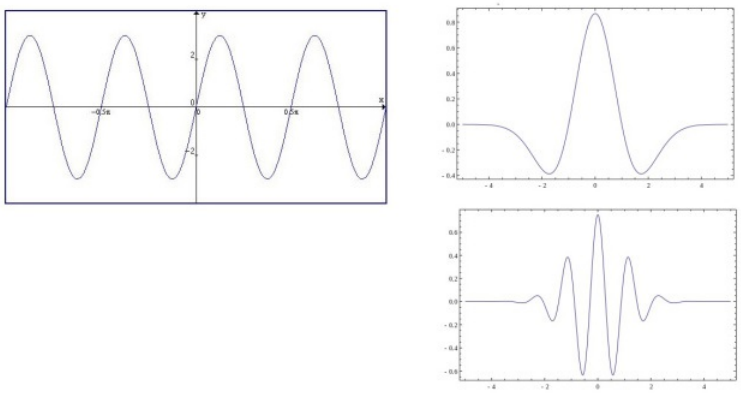
Scopo del presente lavoro è l'implementazione della trasformata discreta di Haar, utilizzando un approccio d'elaborazione basato su parallelismo di natura SIMD - DM.
Nel caso specifico, la trasformata è quella 2D, e gli input/output del problema sono composti essenzialmente da matrici rappresentanti immagini.
Viene quindi utilizzato, per la parte relativa alla visualizzazione e/o manipolazione dei pixel, il framework OpenCV, mentre l'elaborazione della trasformata, che è il cuore del problema, viene svolta utilizzando il framework CUDA, essendo questo tipo di problema particolarmente adatto al calcolo in parallelo, data la natura pixelwise delle operazioni.
Dopo una breve descrizione del problema, verranno trattati i differenti approcci utilizzati (comprendendo anche la versione sequenziale dell'elaborazione, implementata per fornire una misura dei guadagni ottenibili utilizzando il GPGPU computing).
Il lavoro si chiude con i test d'esecuzione, e le misurazioni dei tempi di calcolo al variare degli approcci e delle dimensioni computazionali del problema.

2 Descrizione del problema

Iniziamo con la descrizione di quello che è il problema trattato e dello scenario di utilizzo.

2.1 Wavelets e Trasformata di Haar

Negli ultimi anni un'attenzione sempre crescente è stata dedicata, nel campo informatico e ingegneristico, allo studio delle cosiddette wavelet, relativamente al campo dell'analisi dei segnali e non solo[1].
Rispetto al classico approccio che fa uso della trasformata di Fourier, per rappresentare un segnale nel dominio delle frequenze, il vantaggio principale delle wavelet è quello di mantenere, nel dominio di arrivo, un legame con quello che è il dominio temporale del segnale analizzato, a differenza di quanto avviene con l'analisi di Fourier, cosa che ha sempre rappresentato un'enorme limitazione (nonostante le infinite applicazioni di questo tipo di analisi).
A differenza dell'analisi di Fourier, nella quale un segnale è rappresentato come una serie di funzioni base (di seno e coseno) ciascuna delle quali temporalmente illimitata, le wavelet, ovvero le funzioni base tramite le quali rappresentare una determinata funzione / segnale attraverso una combinazione lineare, sono funzioni temporalmente ben definite e limitate.



a sinistra una funzione seno, a destra 2 wavelets

Continuando il parallelismo con l'analisi di Fourier, nella quale le funzioni base sono combinate variando in frequenza le funzioni seno e coseno, nell'analisi con wavelet, ciascuna di queste è ottenuta come uno scaling (e traslazione) di una funzione "madre" comune. Generalmente il dominio di arrivo

quindi sarà composto dai valori dei parametri di scaling delle wavelet (generate dalla funzione madre), e dai loro coefficienti (usati in combinazione lineare per rappresentare il segnale dato). La funzione definita di seguito è la funzione di Haar, che si adatta abbastanza semplicemente alla generazione di wavelet, ed è per questo considerata come la più antica e semplice tra le wavelet.

Data la funzione madre (funzione di Haar):

$$\psi(x) = \begin{cases} 1 & 0 \leqq x < \frac{1}{2} \\ -1 & \frac{1}{2} \leqq x < 1 \\ 0 & altrimenti \end{cases}$$

le wavelet figlie sono generate dalla seguente espressione:

$$\psi_{j,k}(x) = \psi(2^j x - k)$$

$$\text{con } j \geq 0 \text{ e } 0 \leqq k \leq 2^j - 1$$

dove j è l'indice di scaling, e k quello di traslazione..
Un esempio di generazione delle prime wavelet, a partire dalla funzione madre, è il seguente[2]:

$\psi_{0,0}(x) = \psi(x)$

$\psi_{1,0}(x) = \psi(2x)$

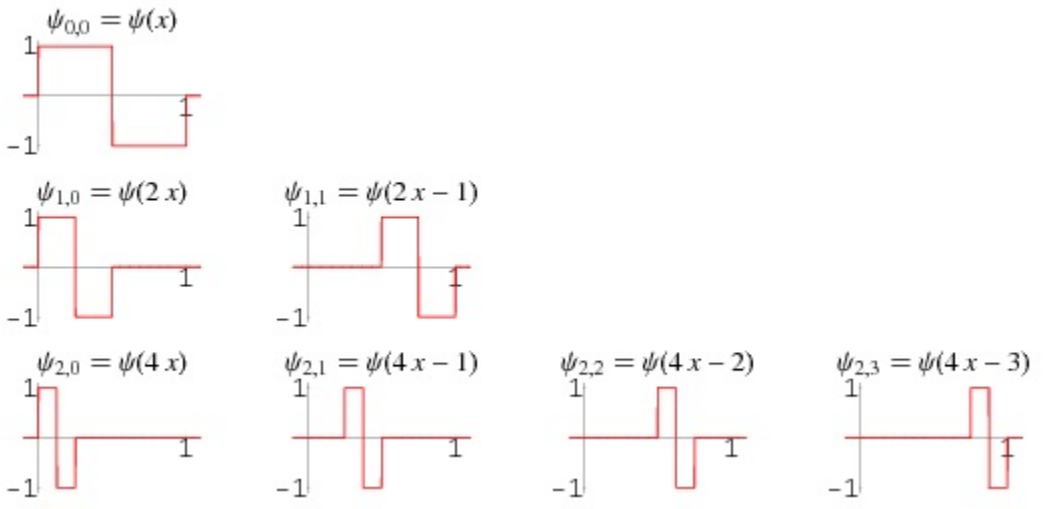
$\psi_{1,1}(x) = \psi(2x - 1)$

$\psi_{2,0}(x) = \psi(4x)$

$\psi_{2,1}(x) = \psi(4x - 1)$

$\psi_{2,2}(x) = \psi(4x - 2)$

$\psi_{2,3}(x) = \psi(4x)$



Quindi una data funzione può essere scritta come combinazione lineare generata dalla seguente formula:

$$f(x) = c_0 + \sum_{j=0}^{\infty} \sum_{k=0}^{2^j-1} c_{j,k} \psi_{j,k}(x)$$

dove $c_{j,k}$sono i coefficienti di Haar, mentre le wavelet $\psi_{j,k}(x)$ sono anche dette basis function, e prese due wavelet qualunque (purchè con coppie di indici differenti tra loro) queste sono funzioni ortogonali, ovvero il loro inner product (l'equivalente del prodotto scalare tra vettori) ottenuto integrando i prodotti puntuali tra le funzioni, vale zero.
In sostanza quindi, fissata una x, esiste una serie di coefficienti che, combinati linearmente con le wavelet generate dalla funzione (madre) di Haar, genera il valore della funzione in x.
Dal punto di vista di funzioni campionate, possiamo utilizzare lo stesso principio.
Se abbiamo un vettore di n elementi (i campioni della nostra funzione) sarà possibile scrivere:

$$\begin{bmatrix} 1 & \sum_{j=0} \sum_{k=0}^{2^j-1} \psi_{j,k}[0] \\ 1 & \sum_{j=0} \sum_{k=0}^{2^j-1} \psi_{j,k}[1] \\ 1 & \sum_{j=0} \sum_{k=0}^{2^j-1} \psi_{j,k}[2] \\ ... & ... \\ 1 & \sum_{j=0} \sum_{k=0}^{2^j-1} \psi_{j,k}[n-1] \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ ... \\ ... \\ c_{n-1} \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ ... \\ ... \\ f_{n-1} \end{bmatrix}$$

dove la matrice quadrata a sinistra (detta **matrice di Haar**) ha in ogni elemento (a partire da quelli in seconda colonna) il valore di una wavelet (quella di indice nella sequenza di quelle generate

dalla funzione madre di Haar uguale all'indice colonna sulla matrice) in uno degli n-intervalli (quello di indice uguale all'indice riga sulla matrice) tra 0 e 1.
Ad esempio una matrice di Haar di ordine 4 ha la forma:

$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & -1 & 0 \\ 1 & -1 & 0 & 1 \\ 1 & -1 & 0 & -1 \end{bmatrix}$$

Si nota quindi il legame tra il dominio temporale (in questo caso rappresentato dall'intervallo del dominio tra 0 e 1, che nella funzione continua era rappresentato da x) fissato scegliendo una riga della matrice, e il coefficiente assegnato a ciascuna wavelet , ognuna con un diverso valore del parametro di scaling..
Si dimostra, ma questo argomento va oltre lo scopo del presente lavoro, che è possibile[4] generare la matrice inversa di quella sopra rappresentata, ovvero la matrice per ottenere i coefficienti a partire dalla funzione campionata (che rappresentano proprio la trasformata di Haar) detta **matrice di Trasformazione di Haar** di ordine n, utilizzando una concatenazione di due matrici, una ottenuta tramite il prodotto di *Kronecker* tra la matrice di trasformazione di Haar di ordine$\frac{n}{2}$ e un vettore [1 1], l'altra ottenuta calcolando il prodotto di Kronecker tra la matrice identità (di ordine $\frac{n}{2}$) ed il vettore [-1 1].
Ad esempio se volessimo calcolare la matrice di trasformazione di Haar di ordine 4 (che ci permette di ottenere i 4 coefficienti di Haar data una funzione costituita da 4 campioni) dovremmo partire dalla matrice di trasformazione di Haar di ordine 2, che è la seguente:

$$H_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

la matrice di trasformazione di Haar di ordine 4 allora sarà data da:

$$H_4 = \begin{bmatrix} H_2 \otimes [1,1] \\ I_n \otimes [1,-1] \end{bmatrix}.$$

Questo dal punto di vista dell'analisi di funzioni utilizzando i coefficienti di Haar.

Tuttavia quello che ci interessa più da vicino, relativamente al nostro lavoro, è il fatto che si possa vedere l'operazione di trasformazione di Haar (quella dell'ottenimento dei coefficienti a partire dalla funzione discreta) come un procedimento iterativo, divisibile in step, tali che ad ogni passo si ottenga una suddivisione del segnale iniziale in coppie di vettori bidimensionali, e si moltiplichi ciascuno di essi per la matrice di trasformazione di Haar di ordine 2. Ciascun prodotto ci fornirà 2 componenti fondamentali, rispettivamente interpretabili come i risultati di un filtraggio low pass e high pass.

2.2 Trasformata discreta di Haar 1D

Iniziamo col mostrare un esempio di utilizzo della trasformata di Haar, relativo al caso monodimensionale, con un esempio che aiuti anche a motivare[3] le scelte e gli approcci implementati.
Supponiamo di voler inviare un vettore **v** composto dai seguenti **n** elementi, o eventualmente una sua approssimazione

$$n = 8$$
$$v^T = \begin{bmatrix} 20 & 25 & 52 & 63 & 38 & 22 & 19 & 14 \end{bmatrix}$$

trasmettendo però un numero minimo di elementi.
Si potrebbe pensare inizialmente di inviare soltanto 4 elementi (vettore **avg)** ciascuno ottenuto come la media tra le coppie del vettore di partenza.

$$avg^T = \begin{bmatrix} 22.5 & 57.5 & 30 & 16.5 \end{bmatrix}$$

Chiameremo questi elementi la *average half*.
Sebbene questa potrebbe essere una buona approssimazione, non ci permette di risalire a quello che è il vettore originale.
Un'alternativa è inviare, assieme ai 4 elementi di sopra, 4 elementi ottenuti come la semidifferenza delle distanze tra i valori (sempre presi in coppia), che chiameremo vettore **diff**, ovvero:

$$diff^T = \begin{bmatrix} 2.5 & 5.5 & -8 & -2.5 \end{bmatrix}$$

Questa porzione è detta la *difference half*.
Il vettore trasmesso, che chiamiamo **y,** quindi è il seguente:

$$y^T = \begin{bmatrix} 22.5 & 57.5 & 30 & 16.5 & 2.5 & 5.5 & -8 & -2.5 \end{bmatrix}$$

Ma in che modo si può ricostruire da quest'ultimo, il vettore iniziale ? Ebbene se associamo ciascun elemento della prima metà, con un elemento della seconda metà (nella posizione corrispondente) otteniamo $\mathbf{n/2}$ coppie ovviamente.

$$(22.5, 2.5)(57.5, 5.5)(30, -8)(16.5, -2.5)$$

Se in ciascuna coppia sottraiamo al secondo elemento, il primo, otteniamo gli elementi del vettore iniziale, di indice pari, mentre sommandoli, otteniamo quelli di indice dispari.
Ma qual è quindi il vantaggio dell'utilizzare quest'approccio, quando si inviano comunque n elementi ? Ebbene il vantaggio si ha quando è possibile arrotondare a 0 (e quindi non inviare) gli elementi *relativamente piccoli* della difference half.
In questo modo, viene inviato un numero di elementi minore di n, e si ottiene un leggero errore nella fase di ricostruzione del vettore originale. Quindi in situazioni in cui esiste un tradeoff accettabile tra errore e compressione, l'uso di questo approccio risulta conveniente.
Immaginiamo ad esempio di trasmettere, una versione arrotondata della *difference part.*

$$\begin{bmatrix} 22.5 & 57.5 & 30 & 16.5 & 0 & 5.5 & -8 & 0 \end{bmatrix}$$

Il vettore ricostruito sarà:

$$\begin{bmatrix} 22.5 & 22.5 & 52 & 63 & 38 & 22 & 16.5 & 16.5 \end{bmatrix}$$

ovvero un'approssimazione, con un certo grado di errore, del vettore originale.
Si può inoltre vedere il processo di ottenimento del vettore **y** come un prodotto righe/colonne tra una matrice $\tilde{W_n}$ ed il vettore v

$$\tilde{W_n} * v = y$$

$$\begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ -\frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -\frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{2} & \frac{1}{2} \end{bmatrix} * \begin{bmatrix} 20 \\ 25 \\ 52 \\ 63 \\ 38 \\ 22 \\ 19 \\ 14 \end{bmatrix} = \begin{bmatrix} 22.5 \\ 57.5 \\ 30 \\ 16.5 \\ 2.5 \\ 5.5 \\ -8 \\ -2.5 \end{bmatrix}$$

mentre la ricostruzione è ottenibile utilizzando la matrice inversa di $\tilde{W_n}$, ovvero $\tilde{W_n^{-1}}$

$$\tilde{W_n^{-1}} * y = v$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 22.5 \\ 57.5 \\ 30 \\ 16.5 \\ 2.5 \\ 5.5 \\ -8 \\ -2.5 \end{bmatrix} = \begin{bmatrix} 20 \\ 25 \\ 52 \\ 63 \\ 38 \\ 22 \\ 19 \\ 14 \end{bmatrix}$$

Si noti inoltre che la matrice per la trasformazione inversa (ovvero per l'ottenimento del vettore originale) è ottenibile moltiplicando per 2 la trasposta della matrice utilizzata per ottenere il vettore da inviare.

$$\tilde{W_n^{-1}} = 2 * \tilde{W_n^T}$$

Da un punto di vista operativo, se la matrice $\tilde{W_n}$ fosse ortogonale (ovvero una matrice nella quale le righe/colonne sono vettori linearmente indipendenti con norma euclidea uguale ad 1) allora varrebbe la proprietà per la quale l'inversa di una matrice coincide con la sua trasposta. Affinchè quindi $\tilde{W_n}$ sia ortogonale, basta moltiplicarla per $\sqrt{2}$.
Infatti se chiamiamo $W_n = \sqrt{2} * \tilde{W_n}$, otteniamo che

$$W_n^{-1} = (\sqrt{2} * \tilde{W_n})^{-1}$$

$$= (1/\sqrt{2})\tilde{W_n^{-1}}$$

$$= (1/\sqrt{2})2\tilde{W_n^T}$$

$$= \sqrt{2}\tilde{W_n^T}$$

$$= (\sqrt{2}\tilde{W_n^T})^T$$

$$= W_n^T$$

La nostra matrice di trasformazione diventa quindi

$$W_n = \begin{bmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \\ -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix}$$

mentre la matrice inversa sarà

$$W_n^{-1} = \begin{bmatrix} \frac{\sqrt{2}}{2} & 0 & 0 & 0 & -\frac{\sqrt{2}}{2} & 0 & 0 & 0 \\ \frac{\sqrt{2}}{2} & 0 & 0 & 0 & \frac{\sqrt{2}}{2} & 0 & 0 & 0 \\ 0 & \frac{\sqrt{2}}{2} & 0 & 0 & 0 & -\frac{\sqrt{2}}{2} & 0 & 0 \\ 0 & \frac{\sqrt{2}}{2} & 0 & 0 & 0 & \frac{\sqrt{2}}{2} & 0 & 0 \\ 0 & 0 & \frac{\sqrt{2}}{2} & 0 & 0 & 0 & -\frac{\sqrt{2}}{2} & 0 \\ 0 & 0 & \frac{\sqrt{2}}{2} & 0 & 0 & 0 & \frac{\sqrt{2}}{2} & 0 \\ 0 & 0 & 0 & \frac{\sqrt{2}}{2} & 0 & 0 & 0 & -\frac{\sqrt{2}}{2} \\ 0 & 0 & 0 & \frac{\sqrt{2}}{2} & 0 & 0 & 0 & \frac{\sqrt{2}}{2} \end{bmatrix}$$

Si noti che eseguire il prodotto $W_n v$ equivale a :

- far scorrere un kernel di 2 elementi $\left[\frac{\sqrt{2}}{2} \quad , \frac{\sqrt{2}}{2}\right]$ su tutto il vettore v, centrandolo di volta in volta su ciascuna delle coppie, ed eseguendo una convoluzione. Questo produce un sottovettore, che è proprio quello che abbiamo chiamato **avg**, ovvero la average part, che ha dimensione n/2, interpretabile come il sottocampionamento (di dimensione 2) del vettore originale, dove ogni suo elemento equivale alla media (moltiplicata per $\sqrt{2}$) di 2 elementi adiacenti di **v.** Questo è proprio un processo di sottocampionamento e di smoothing (filtraggio passa-basso). Questo tipo di kernel restituisce un valore vicino a 0 per 2 elementi adiacenti di valore circa opposto, mentre fornisce un valore vicino alla media (per il fattore di cui sopra) per elementi di valore simile.

- far scorrere un kernel, sempre di 2 elementi, $\left[+\frac{\sqrt{2}}{2} \quad , -\frac{\sqrt{2}}{2}\right]$ su v, centrando sempre su ciascuna coppia di elementi adiacenti. La convoluzione con questo kernel, produce la seconda parte del vettore **y,** ovvero la difference part. Anch'essa interpretabile come un sottocampionamento del vettore originale **v**, nel quale pero' ciascun elemento è la semidifferenza (moltiplicata per un fattore $\sqrt{2}$) tra 2 elementi consecutivi. Questo procedimento equivale ad un sottocampionamento con filtraggio passa-alto. Questo kernel restituisce valori vicini allo zero per elementi adiacenti con valori simili, e restituisce valori vicini alla media per valori opposti, fornendo quindi un output che evidenzia le zone con maggiori variazioni di valore.

I kernel $\left[\frac{\sqrt{2}}{2} \quad , \frac{\sqrt{2}}{2}\right]$ e $\left[+\frac{\sqrt{2}}{2} \quad , -\frac{\sqrt{2}}{2}\right]$ sono appunto i filtri della trasformata discreta di Haar, e le matrici quadrate W_ne W_n^{-1}sono le matrici utilizzate per ottenere la trasformata di Haar e la trasformata inversa.
L'utilizzo che si fa della trasformata di Haar, prevede che generalmente si iteri il procedimento sull'average part, che diviene il nuovo vettore in input. In tal modo si può quindi reiterare anche il procedimento di ricostruzione inversa.

2.3 Trasformata discreta di Haar 2D

Illustriamo ora il caso della trasformata di Haar applicata a matrici bidimensionali[3], invece che a vettori.
Nello specifico, si tratta di un cosiddetto processo con kernel separabile, ovvero è possibile implementare una trasformata di Haar 2D eseguendo 2 trasformate di Haar 1D, prima lungo le colonne e poi lungo le righe della matrice di input.
Data quindi una matrice A_{mn} iniziamo con l'eseguire il prodotto righe/colonne

$$B_{mn} = W_m * A_{mn}$$

Ciascun elemento $B(i,j)$ sarà ottenuto col prodotto scalare tra la riga i-esima della matrice di Haar W_me la colonna j-esima della matrice in input. Tuttavia, fissando una colonna k-esima (con k tra 0 e n-1) della matrice di output B_{mn}, e scorrendo lungo le m righe, individuiamo il vettore ottenuto come prodotto tra la matrice di Haar W_me la colonna k-esima della matrice di input A_{mn}.
Questo vuol dire che si può vedere la matrice B_{mn} come formata da tutte le colonne della matrice di input A_{mn} dopo essere state sottoposte, ciascuna, alla trasformata di Haar 1D. Ne consegue che A_{mn} presenta nella metà superiore (formata da m/2 righe) le average parts, e nella metà inferiore

le difference parts.

A questo punto, per terminare la trasformazione 2D, data la separabilità del kernel di trasformazione, dobbiamo trasformare B_{mn} lungo le righe.

Facciamo questo tramite il prodotto

$$C_{mn} = B_{mn} * W_n^T$$

W_n^T non è altro che la matrice per la trasformata discreta di Haar 1D, di dimensione n x n, trasposta in modo tale che, previa post-moltiplicazione con la matrice ottenuta al passo precedente, si ottenga una matrice C_{mn} nella quale ciascun elemento $C(i, j)$ equivale al prodotto scalare tra la riga i-esima della matrice B_{mn} , e la colonna j-esima di W_n^T (che altro non è che una riga, trasposta, della classica matrice per la trasformata di Haar, W_n).

Ed infatti, fissando una riga k-esima della matrice di output finale C_{mn} , e muovendoci lungo le colonne, stiamo visitando il vettore riga ottenuto moltiplicando la riga k-esima di B_{mn} per l'intera matrice W_n^T, vettore che altro non è che la trasformata di Haar della riga k-esima di B_{mn} .

Come è intuibile, il risultato del secondo passo della trasformata 2D avrà le average parts (di B_{mn}) nel quadrante verticale di sinistra, e le difference parts in quello di destra. Ovviamente, essendo B_{mn} già divisa in un quadrante superiore ed uno inferiore, C_{mn} presenterà quattro quadranti. Quello in alto a sinistra, contenente le averages lungo le colonne e poi lungo le righe. Questo quadrante rappresenta un sottocampionamento nelle 2 dimensioni, dell'immagine, con un filtraggio passa basso in entrambe le direzioni.

Quello in alto a destra contenente prima le averages lungo le colonne e poi le differences lungo le righe. Tale quadrante equivale ad un sottocampionamento della matrice originale nelle due direzioni, con un filtraggio passa basso lungo le colonne, ed un filtraggio passa alto lungo le righe.

Quello in basso a sinistra contenente le differences lungo le colonne e poi le averages lungo le righe, quindi un passa alto lungo le colonne ed un passa basso lungo le colonne, ed infine quello in basso a destra contenente le differences sia lungo le colonne che lungo le righe, quindi sottocampionamento e filtraggio passa alto in entrambe le direzioni.

Riassumendo, data una matrice A_{mn} , la sua trasformata discreta di Haar C_{mn} è ottenibile tramite

$$C_{mn} = W_m A_{mn} W_n^T$$

Allo stesso modo che nel caso 1D, anche per la trasformata 2D di Haar, si itera il procedimento facendo si che la nuova matrice di input sia il quadrante contenente le medie in entrambe le direzioni

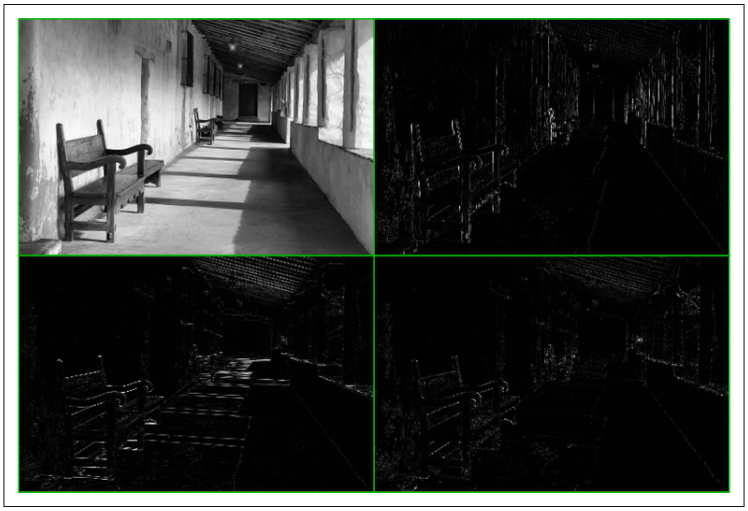
(il quadrante in alto a sinistra) ottenuto con la trasformata 2D di Haar al livello precedente (quindi la nuova dimensione computazionale è ridotta di un fattore 4).

2.4 Applicazione della trasformata discreta 2D di Haar alle immagini

Se applichiamo quanto visto per la trasformata 2D di Haar, ad una matrice rappresentante un'immagine (gli elementi i,j rappresentano i pixel di indice riga i ed indice colonna j) possiamo più facilmente intuire il funzionamento di tale procedimento.
Immaginiamo di avere in input l'immagine

Dopo il primo passo, ovvero l'applicazione della trasformata di Haar a ciascuna colonna, otterremo:

Applicando a ciascuna riga di questa, la trasformata di Haar, otterremo:



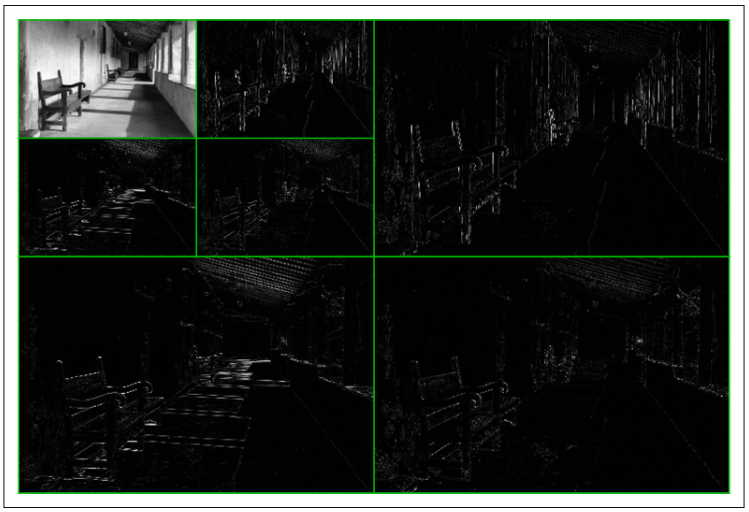
Osservando i quattro quadranti di cui è composta l'immagine, notiamo che quello in alto a sinistra è una versione sottocampionata e *smussata,* mentre quella in basso a destra presenta valori diversi da zero in corrispondenza dei pixel per i quali (nell'immagine originale) erano presenti variazioni d'intensità notevoli, in entrambe le direzioni. Per il quadrante in alto a destra, sono evidenziate le variazioni d'intensità lungo la direzione orizzontale, mentre per il quadrante in basso a sinistra quelle lungo la direzione verticale.

Come già detto inoltre, ciascuno dei 4 quadranti è una versione sottocampionata dell'immagine di partenza. Ovvero data un'immagine di dimensioni mxn, ciascun quadrante è un'immagine di dimensione $\frac{m}{2}x\frac{n}{2}$, in quanto ogni pixel di output è ottenuto come combinazione lineare di un intorno di 4 pixel (sottocampionamento nelle due dimensioni) dell'immagine di input. I coefficienti di questa combinazione lineare sono dati dall'utilizzo in sequenza dei 2 filtri di Haar.

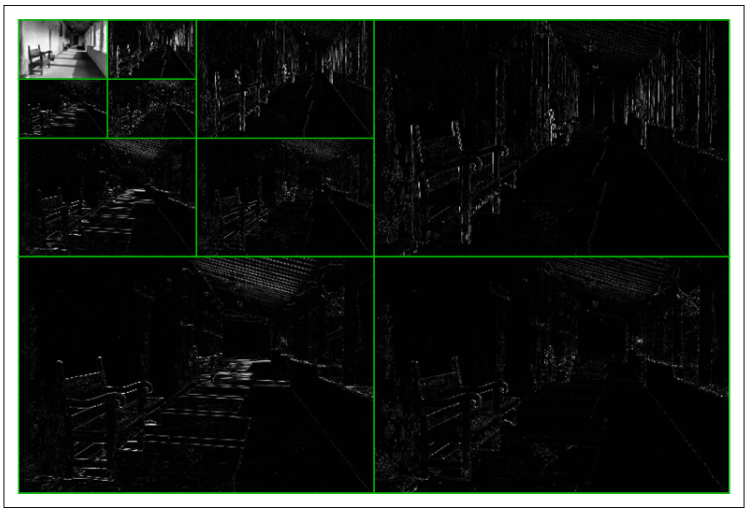
Infatti mentre nell'eseguire la trasformata di Haar 1D di un vettore, si applica **sia** il filtro passa-basso di Haar $\left[\frac{\sqrt{2}}{2} \quad , \frac{\sqrt{2}}{2}\right]$, **sia** quello passa-alto $\left[\frac{\sqrt{2}}{2} \quad , -\frac{\sqrt{2}}{2}\right]$, ottenendo un unico vettore composto da 2 versioni sottocampionate (e filtrate) di quello di partenza, e così come nell'eseguire la trasformata 2D di un'immagine (matrice) si applicano sia i filtri passa-basso che passa alto di Haar, sia lungo le colonne sia lungo le righe, se invece consideriamo ciascuno dei quattro quadranti singolarmente, ognuno di essi è ottenuto quindi tramite (oltre al sottocampionamento) l'applicazione o del passa-basso o del passa-alto lungo le colonne, ed equivalentemente lungo le righe.

A seconda quindi del quadrante di appartenenza, del pixel dell'immagine di output, è possibile non solo individuare i 4 pixel dell'immagine originale dai quali è stato ricavato, ma anche quelli che sono i coefficienti di un kernel 2x2 di convoluzione, applicato a tale intorno di 4 pixel. In tal modo

si può implementare la trasformata 2D di Haar (e la trasformazione inversa) come un'operazione interamente pixel-wise che sfrutti quindi appieno i vantaggi del calcolo parallelo.
Iterando il procedimento sul quadrante contenente le medie nelle due direzioni, otteniamo:



e ancora

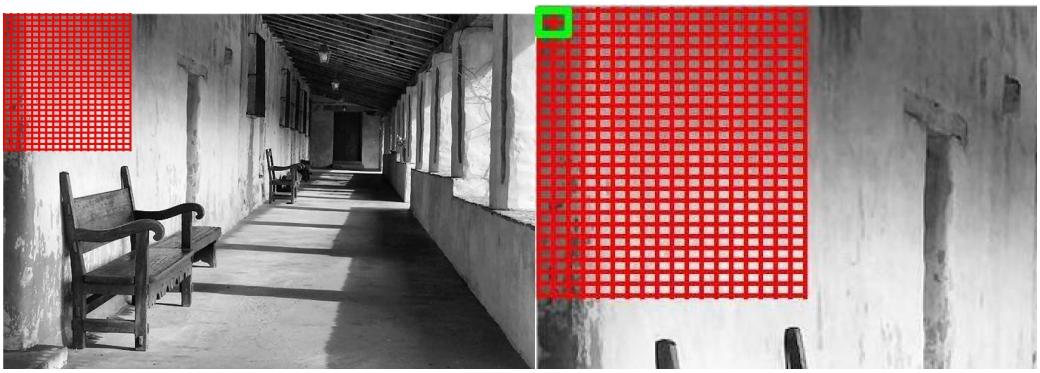


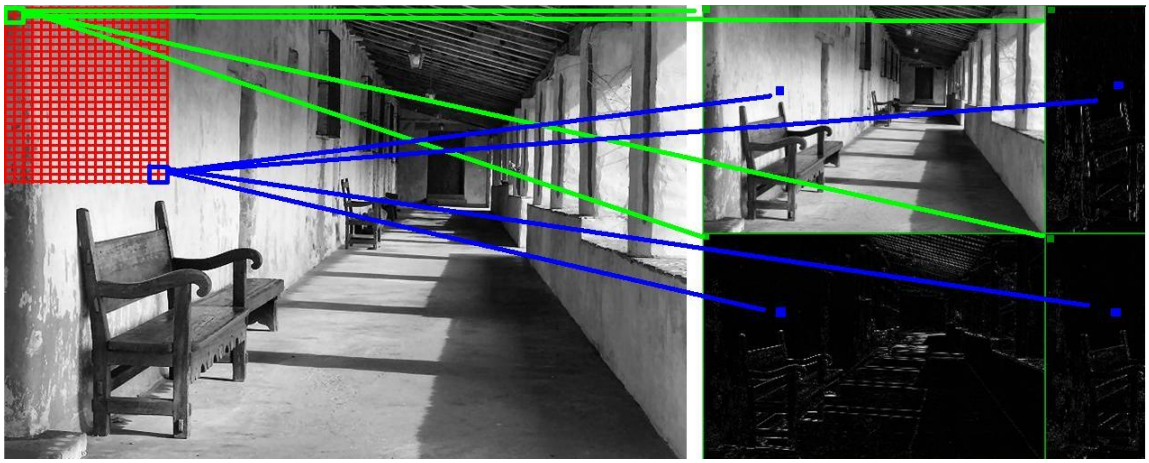
3 Approcci utilizzati

Illustriamo adesso gli approcci scelti per calcolare la trasformata di Haar, sia per la versione sequenziale, sia per la versione parallela..

3.1 Mappatura pixel in input e pixel in output

Riprendiamo quanto detto nel capitolo precedente, relativamente alla suddivisione in quattro quadranti dell'immagine di output. Data un'immagine in input, ciascuno dei 4 sottoquadranti dell'immagine ottenuta come trasformazione di Haar (iterata una sola singola volta), è una versione sottocampionata dell'immagine originale (sottocampionamento di dimensione 2 prima lungo le colonne e poi lungo le righe). In tal modo, data un'immagine mxn, otterremo una trasformata composta da 4 quadranti, ciascuno di dimensione $\frac{m}{2}x\frac{n}{2}$ pixels. Ciascuno dei pixel (di ciascun quadrante) è ottenuto come combinazione lineare di un gruppo di 4 pixel adiacenti, nell'immagine di input.
Più formalmente, se dividiamo l'immagine di input in gruppi di $2x2$ elementi, e assegnamo a ciascuno di questi blocchi un indice riga $\bar{I}$ ed un indice colonna $\bar{J}$,dato un pixel dell'immagine trasformata $p(i,j)$, questo è calcolato usando i 4 pixel appartenenti al blocco di indici $\bar{I} = i\,\%(m/2)$ e $\bar{J} = j\,\%(n/2)$.
Nell'immagine seguente, la griglia rossa rappresenta i pixel dell'immagine da trasformare, mentre in quella successiva, è mostrata la corrispondenza tra i 4 pixel (di 4 quadranti) dell'immagine trasformata, e il gruppo di 4 pixel dell'immagine originale (questo per 2 distinti gruppi di 4, il verde ed il blu).





Quindi, se chiamiamo

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

i 4 pixel di un intorno 2x2 dell'immagine di input, sarà possibile calcolare i pixel della trasformata con le seguenti combinazioni lineari:

- $\frac{\sqrt{2}}{2}(\frac{\sqrt{2}}{2}a + \frac{\sqrt{2}}{2}b) + \frac{\sqrt{2}}{2}(\frac{\sqrt{2}}{2}a + \frac{\sqrt{2}}{2}b) = \frac{1}{2}(a+b+c+d)$ per pixel del quadrante in alto a sinistra nell'immagine di output
- $\frac{1}{2}(b+d-a-c)$ per pixel del quadrante in alto a destra
- $\frac{1}{2}(d+c-b-a)$ per il quadrante in basso a sinistra
- $\frac{1}{2}(d-b-c+a)$ per il quadrante in basso a destra

Tuttavia vedremo come, nell'implementazione in CUDA, le cose si complicheranno leggermente, sia perchè non lavoreremo con matrici bidimensionali (ma con gli array monodimensionali che le rappresentano) sia perchè prevederemo un tipo di disposizione dei dati tale da permettere la cosiddetta lettura coalescente.

3.2 CUDA e la coalescenza

Gli approcci paralleli, utilizzati per ottenere la trasformata di Haar di un'immagine, hanno il loro cuore computazionale nell'utilizzo del framework CUDA C.

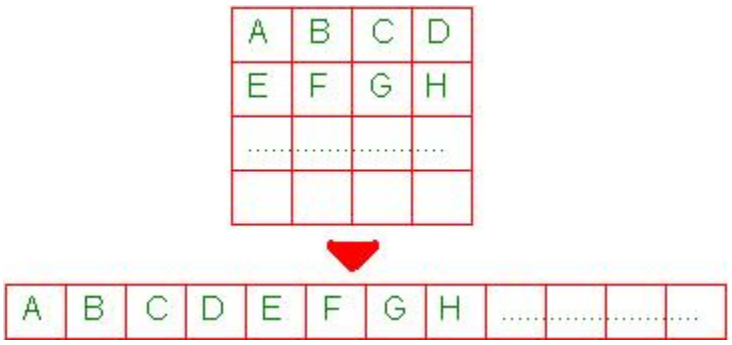
CUDA è un framework per il calcolo parallelo su GPU, che permette quindi di sfruttare l'enorme capacità computazionale delle moderne schede video, evitando al programmatore l'onere di dover lavorare con primitive proprie della pipeline grafica.
Con CUDA quindi è possibile modellare algoritmi basati sul paradigma SIMD-DM, ovvero che vedono un grande numero di thread impegnati contemporaneamente tutti nello stesso tipo di operazione, su dati generalmente condivisi.
Senza scendere nel dettaglio del meccanismo di lavoro della libreria, limitiamoci a dire che un codice da eseguire in parallelo è descrivibile tramite un particolare tipo di routine, detta *kernel*, che viene eseguito sulla matrice di unità d'elaborazione delle GPU, a seconda di quella che è la configurazione scelta dal programmatore all'avvio del kernel.
Un kernel è descritto tramite una configurazione, detta *griglia* di thread suddivisi in blocchi.
Il mapping tra la griglia e la matrice di processori è eseguita in maniera trasparente per il programmatore.
L'hardware GPU utilizza una memoria totalmente separata da quella classica utilizzata nei normali programmi eseguiti su CPU, ed infatti tutti i dati da elaborare in GPU, devono essere preventivamente copiati sulla sua memoria (e ricopiati nella memoria accessibile dalla CPU una volta che l'elaborazione è completata).
Quello che ci interessa principalmente, per quanto concerne il qui presente lavoro, sono 2 principi dell'uso della memoria in CUDA:

1. Tutti i thread della griglia possono accedere alla stessa memoria *globale* della GPU. Tuttavia è possibile allocare spazi di memoria locali a ciascun blocco, visibili solo dai thread locali al blocco. Per questi spazi di memoria *condivisa* (dai thread del blocco) i tempi di accesso sono molto minori rispetto a quelli per la memoria globale della GPU.

2. Thread con id consecutivi, che accedono a parole di memoria, di grandezza multipla di 4 byte, consecutive in memoria globale, partendo da un'indirizzo base che è multiplo della granularità (dimensione) delle parole, eseguono un accesso detto *coalescente* . Quando un accesso è coalescente, con un solo accesso alla memoria vengono recuperate tutte le parole di memoria lette da un sottogruppo di 16 thread detto half-warp (nelle versioni più recenti dell'hardware abbiamo un intero warp di 32 thread). Le letture coalescenti quindi risultano molto più veloci di quelle non coalescenti. La lettura coalescente è ottenuta anche se alcuni thread non leggono alcun dato, purchè sia mantenuto un allineamento uguale, per i thread che accedono in memoria, a quello che si avrebbe se tutti i thread leggessero (quindi senza

incroci).

Vogliamo quindi far sì che i thread di un blocco sfruttino quanto più possibile la memoria condivisa, e vogliamo inoltre che le letture in memoria globale siano coalescenti.
Ma come si applicano, queste considerazioni, al nostro problema ?
Innanzitutto iniziamo col dire che anche quando si lavora con matrici 2D, in CUDA, si fa in modo da avere in memoria sempre un vettore monodimensionale che rappresenti la matrice data.
Mentre questo risulta automaticamente vero in C per matrici allocate staticamente (su stack) in quanto in memoria risultano come un unico vettore ottenuto concatenando le righe della matrice (allocazione per righe del C), quando allochiamo dinamicamente (su heap) una matrice 2D, non possiamo utilizzare il classico approccio dell'array di puntatori a puntatori (sebbene anche le matrici 2D statiche siano interpretabili come puntatori ad array di puntatori) in quanto l'allocazione dinamica delle righe non fornirebbe zone di memoria contigue. Si preferisce quindi lavorare allocando vettori monodimensionali di size uguale al totale degli elementi della matrice 2D, e mappando gli accessi con la classica formula che ci dice che accedere ad un elemento (i,j) di una matrice, allocata usando un array monodimensionale contiguo che la rappresenta, equivale ad accedere all'elemento $i * n + j$.



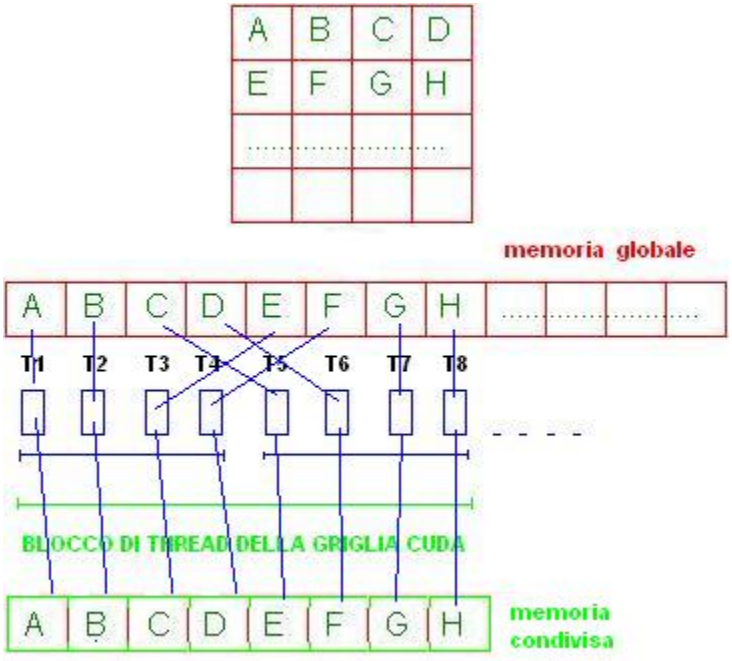
L'immagine sopra mostra il modo in cui una matrice 2D è allocata in memoria.
In accordo a quanto precedentemente illustrato per la trasformata di Haar 2D, sappiamo che i 4 elementi (pixel) A,B,E,F ,nella matrice dell'immagine di input, contribuiscono al calcolo di 4 pixel differenti (simmetricamente disposti in ciascuno dei 4 sottoblocchi dell'immagine trasformata). Ha senso avere quindi un thread per ciascun pixel dell'immagine di input ($m * n$ thread totali quindi).
.

Tutti i thread sono logicamente divisi (qui si parla di gruppi logici di lavoro, non dei blocchi della griglia CUDA) in gruppi di 4 thread, che condividono i 4 pixel (dell'immagine di input) necessari, a ciascuno dei thread, per calcolare il corrispondente pixel nell'immagine di output. .

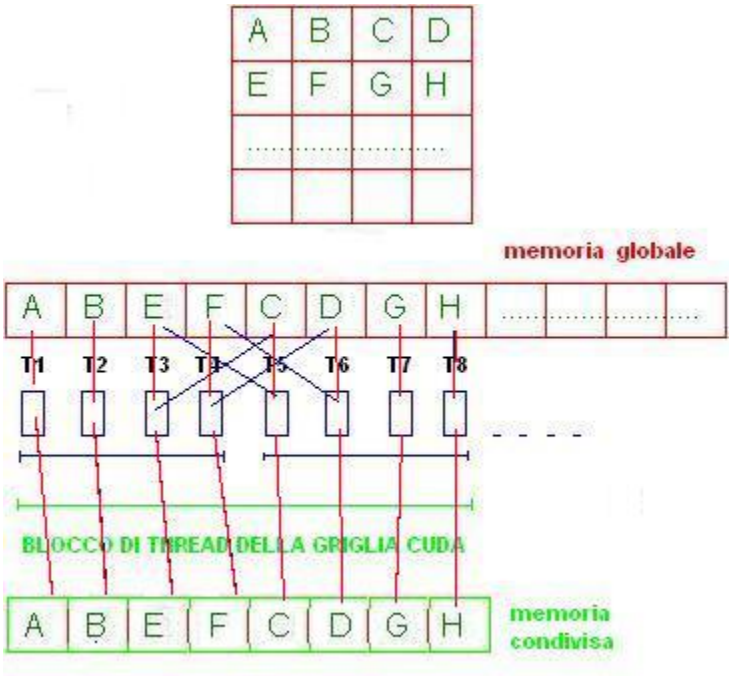
Il fatto che 4 thread debbano accedere agli stessi 4 pixel dell'immagine di input, fa immediatamente pensare alle memoria condivisa. .

Si può quindi prevedere che ciascun thread dei 4 copi in memoria condivisa (leggendo dalla memoria globale) un solo pixel dei 4, e poi utilizzi anche gli altri 3, copiati in memoria condivisa dagli altri thread del gruppo di lavoro logico (questo necessita ovviamente di una configurazione della griglia per la quale il numero di thread appartenenti al blocco sia multiplo intero di 4, per evitare di avere gruppi di lavoro logici incompleti, cioè con meno di 4 thread).

Avremmo quindi la seguente situazione:



Tuttavia, come si nota dall'immagine sopra, in questo modo non otterremmo una lettura coalescente. Quello che ci interessa ottenere quindi, è una disposizione di memoria dei pixel di questo tipo:



Prevederemo quindi una fase iniziale di ridisposizione della matrice di input, che diventerà l'input dell'algoritmo che effettua la trasformata.

Sia questa ridisposizione, sia la trasformata di Haar, verranno implementate tramite 2 kernel CUDA separati, e verranno fornite 2 differenti versioni per entrambi i processi: una che esegue la ridisposizione e la trasformazione di Haar per ogni successivo livello di nesting del procedimento, copiando di volta in volta la matrice dell'immagine (o del sottoquadrante dell'immagine) da elaborare, e i risultati ottenuti, dalla memoria host (quella alla quale la CPU ha accesso) alla memoria device (GPU) (e viceversa) ed una che esegue le chiamate ai kernel, per successivi livelli di nesting della trasformata di Haar, in modo tale che questi lavorino su dati residenti in memoria device fino alla fine del processo di ottenimento di tutti i livelli di nesting. E vedremo che è proprio in quest'ultimo caso che si ottengono le migliori prestazioni, essendo il trasferimento tra memoria host e memoria device, nel GPU Computing, il principale collo di bottiglia dell'elaborazione.

Iniziamo la trattazione analizzando l'implementazione della trasformata di Haar effettuata in sequenziale sulla CPU, e proseguiamo illustrando le due strategia sopra anticipate.

3.3 Trasformata di Haar dell'immagine in sequenziale

Utilizzando la separabilità della trasformata 2D di Haar, otteniamo il risultato ricercato applicando una trasformata di Haar prima a ciascuna colonna, e poi a ciascuna riga dell'output ottenuto lavorando sulle colonne.
Abbiamo già detto che il primo step della trasformata 2D di Haar consiste nel prodotto

$$B_{mn} = W_m * A_{mn}$$

il cui risultato ci da un'immagine delle stesse dimensioni di quella originale, dove ciascuna colonna è formata da 2 parti, quella superiore delle medie e quella inferiore delle semidifferenze. Ciascuna di queste 2 parti è una versione sottocampionata (quindi di dimensione dimezzata) della colonna originale. Possiamo quindi ottenere questo tipo di trasformazione immaginando di scorrere sui pixel dell'immagine di output. Per ciascuna colonna dell'output, possiamo scorrere su ciascuna riga, e a seconda della riga (prima o dopo $\frac{m}{2}$ dove m è il numero di righe) utilizzare uno dei 2 filtri (low-pass o high-pass) di haar. Il primo dei 2 pixel dell'immagine di input, che contribuiscono al calcolo del valore del pixel di output (sul quale siamo posizionati) sui quali cioè stiamo centrando i filtri di Haar, avrà indice di colonna uguale a quello del pixel di output, e indice riga ottenibile come il doppio dell'indice di riga del pixel di output modulo m.
In questo modo, tutti i pixel di una colonna, vengono associati in coppia, e ciascuna coppia è utilizzata 2 volte (una volta per il calcolo della prima metà del vettore di output , quella ottenuta col filtro passa-basso, e una volta per il calcolo della seconda metà).
In pseudocodice l'algoritmo per il calcolo della trasformata di Haar delle colonne dell'immagine è il seguente:

```
n=colonne
m=righe
k=0;
for j in [0,n-1]
    k=0;
    for i in [0,m-1]
        if (i< m/2 )
            out[i,j] = coeff *( in[k%m,j] + in[(k+1)%m, j];
        else
            out[i,j] = coeff * (-in[k%m,j] + in[(k+1)%m,j];
        k=k+2;
```

ed equivalentemente per la trasformata delle righe, ovvero il prodotto

$$C_{mn} = B_{mn} * W_n^T$$

l'algoritmo è il seguente (si noti che in questo caso in[] è l'immagine ottenuta al primo step).

```
n=colonne
m=righe
k=0;
for i in [0,m-1]
    k=0;
    for j in [0,n-1]
        if (j< n/2 )
            out[i,j] = coeff *( in[i,k%n] + in[i,(k+1)%n];
        else
            out[i,j] = coeff * (-in[i,k%n] + in[i,(k+1)%n];
        k=k+2;
```

Tale algoritmo ottiene la trasformata di Haar, di un livello, dell'immagine fornita in input. Per ottenere livelli successivi della trasformata, basta reiterare il procedimento, scegliendo per come nuova immagine di input il quadrante in alto a sinistra della trasformata ottenuta al passo percedente.

3.4 Trasformata di Haar dell'immagine in parallelo, con trasferimenti multipli tra le memorie

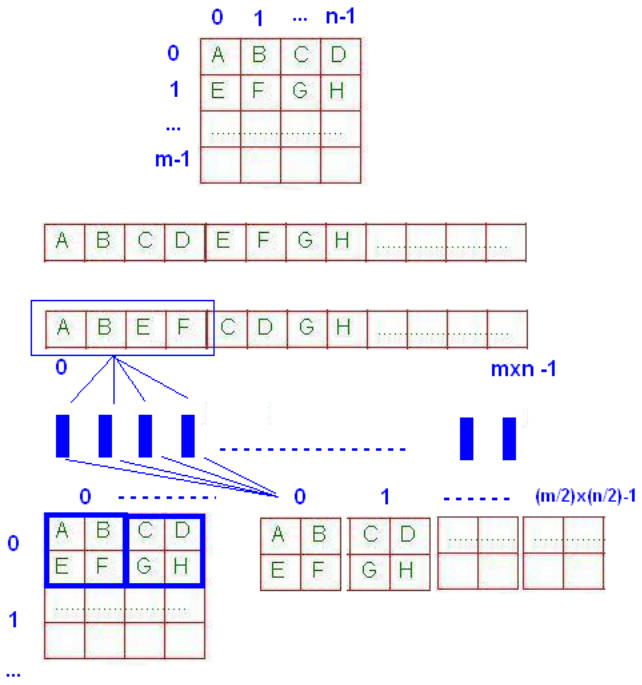
In questo tipo di approccio vogliamo utilizzare il parallelismo offerto dall'ambiente CUDA e dal GPU computing, utilizzando i principi visti prima per la lettura coalescente e l'utilizzo della shared memory.

In questo approccio, ciascun livello di nesting della trasformata è trattato come una procedimento indipendente. Per k livelli quindi , verra reiterato k volte lo stesso approccio utilizzato per un singolo livello.

I passi principali prevedono la configurazione di una griglia, e la chiamata ad un kernel CUDA nel quale più thread ridispongono i dati della matrice in input in modo tale che la successiva trasformazione effettui una lettura coalescente, e la configurazione e chiamata di un kernel per la trasformata di Haar 2D. Lasciando per ora da parte gli aspetti implementativi, e le routine CUDA utilizzate per gestire gli aspetti legati alla memoria device, concentriamoci sull'aspetto logico dei 2 kernel usati.

Abbiamo detto che a ciascun livello di nesting della trasformata di Haar, secondo questo tipo di approccio, l'input del problema è una matrice rappresentante un'immagine, nella forma classica (allocazione per righe) e l'output intermedio dovrà essere una matrice disposta (linearmente) secondo l'ordine sopra illustrato per garantire la lettura coalescente. L'output finale sarà la trasformata ottenuta sull'output intermedio di pixel ridisposti.

Iniziamo a mostrare con un'immagine quella che è la logica utilizzata nella fase di **ridisposizione**:



La prima matrice nell'immagine è la nostra immagine in input. .

Il primo vettore monodimensionale è il modo in cui questa appare in memoria, mentre il secondo vettore è come vogliamo che sia disposta.

Le barre blu rappresentano i thread appartenenti alla griglia CUDA. Siccome avremo un thread per ciascuno degli mxn pixel dell'immagine, gli id dei thread, relativamente alla griglia (un thread ha sia id relativo al blocco di thread CUDA che all'intera griglia) è possibile calcolare il quadrante di pixel relativo ad un thread (nella disposizione lineare che vogliamo ottenere) come *(id globale del thread) diviso 4*. Più precisamente questo sarà l'id (tra 0 ed $\frac{m}{2}x\frac{n}{2} - 1$) del blocco di 4 pixel (nelle formule *idQuad*), in un immaginario array monodimensionale in cui si potrebbero disporre i blocchi di 4 pixel dell'immagine, per come si incontrano allocando l'immagine per righe.

A questo punto è possibile ottenere l'indice riga (I) e l'indice colonna (J)del blocco di 4 thread, nell'immagine idealmente suddivisa in tali blocchi da 4 (ultima matrice in basso a sinistra dell'immagine). Tali indici sono calcolabili rispettivamente come indice del blocco di 4 pixel (calcolato sopra) diviso$\frac{n}{2}$e modulo $\frac{n}{2}$. Ed inoltre, è possibile calcolare l'indice locale (nelle formule successive *indInSottogruppo*) di un thread all'interno del gruppo logico di 4 threads assegnati allo stesso blocco di 4 pixel, come (id globale del thread) modulo 4. In ultima istanza, passando da quest'ultimo, che

è un indice lineare, ad una disposizione 2D, possiamo calcolare l'indice riga/colonna che il thread occupa all'interno del gruppo di 4 thread, disposti come un quadrato 2x2, dividendo rispettivamente l'indice locale del thread nel gruppo logico di 4 per 2 (per l'indice riga, *iSottogruppo* nella formula) e calcolandone il modulo 2 (per l'indice colonna, *jSottogruppo*).

$idQuad = \frac{globId}{4}$

$indInSottogruppo = globId\%4$

$\bar{I} = \frac{idQuad}{(n/2)}$, $\bar{J} = idQuad\%\frac{n}{2}$

$iSottogruppo = \frac{indInSottogruppo}{2}$

$jSottogruppo = indInSottogruppo\%2$

Ricordiamo che l'indice globale del thread nella griglia è calcolabile come l'id del blocco CUDA moltiplicato il numero di thread del blocco più l'id del thread localmente al blocco CUDA (si faccia attenzione a distinguere il blocco CUDA che è relativo alla struttura e alla configurazione della griglia CUDA, da quello che chiamo il blocco logico di lavoro, ovvero 4 thread consecutivi che lavorano con gli stessi 4 pixels).
A questo punto ciascun thread, può posizionare in memoria globale, in indice pari al suo id globale nella griglia CUDA, il pixel dell'immagine che si trova all'indice:

$$(2 * \bar{I} + iSottogruppo) * n + 2 * \bar{J} + jSottogruppo$$

e questo ci darà la disposizione richiesta per la successiva lettura coalescente nel kernel che si occuperà dell'effettivo calcolo della trasformata di Haar. .
A questo punto quindi possiamo lanciare il kernel che si occupa della trasformata di Haar, lavorando direttamente sullo spazio di memoria globale (del device) su cui abbiamo disposto i pixel dell'immagine secondo l'ordine richiesto..
Concettualmente, ciascun thread (1 per pixel di output) ha bisogno di 4 pixel di input per calcolare il valore di output corrispondente. Tuttavia, volendo utilizzare la memoria condivisa, tra i thread di un blocco, ed essendo un blocco composto da un numero di thread multiplo di 4, basterà che ciascun thread copi uno solo dei 4 pixel del blocco logico di lavoro (di 4 thread-4pixel) a cui è

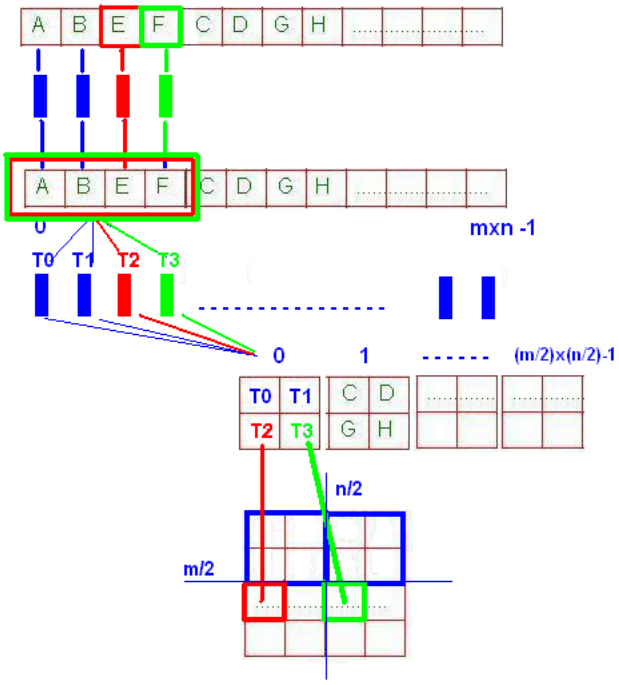
assegnato, dalla memoria globale alla memoria condivisa. Il calcolo del valore del pixel di output verrà eseguito utilizzando i 4 pixel contenuti nella shared memory. .
Equivalentemente a quanto effettuato nel kernel che si occupava di ridisporre i pixel, anche in questo kernel calcoliamo:

- l'id globale del thread nella griglia
- l'indice lineare del gruppo di 4 pixel a cui tale thread è assegnato nell'immagine di input
- l'indice riga e l'indice colonna del gruppo di 4 pixel per come questo appare sull'immagine in input originale
- l'indice (tra 0 e 3) del thread all'interno del gruppo logico di lavoro di 4 thread assegnati ai 4 pixel
- gli indici riga e colonna (tra 0 e 1) del thread all'interno del gruppo logico di lavoro di 4 thread, disposti in un quadrato di dimensione 2x2

A seconda del valore dell'indice lineare del thread all'interno del gruppo logico, è possibile risalire alla posizione (nei 4 sottoquadranti finali dell'immagine trasformata) del pixel di output calcolato da quel dato thread, e di conseguenza ai coefficienti per i quali tale thread deve moltiplicare ciascuno dei 4 pixel del gruppo logico per ottenere il pixel trasformato. Tutto ciò è fatto in parallelo dai thread della griglia CUDA, dopo una fase iniziale (a monte di una barriera di sincronizzazione) di copia dei pixel da elaborare dalla memoria globale (in cui erano stati messi dal kernel di ridisposizione dei pixel) alla memoria condivisa..
L'indice (nel vettore monodimensionale che rappresenta l'immagine di output, la nostra trasformata di Haar 2D) in cui andrà posizionato il pixel di output, calcolato da ciascun thread, è il seguente:

$$\bar{I} * n + iSottogruppo * \frac{m}{2} * n + \bar{J} + jSottogruppo * \frac{n}{2}$$

Tale dinamica è illustrata nell'immagine seguente:



I quattro thread appartengono allo stesso gruppo logico di lavoro da 4, e quindi anche allo stesso blocco CUDA. Nello specifico vengono mostrate (nell'ultima matrice in basso) i pixel di output dei thread T2 e T3.

Come già detto precedentemente questi 2 kernel implementano un singolo livello della trasformata di Haar.

Prima di chiamare il primo dei 2 kernel (quello di ridisposizione) occorre copiare in memoria device l'array monodimensionale che rappresenta l'immagine da trasformare (per livelli successivi si passa il quadrante in alto a sinistra dell'ultima trasformata ottenuta) e dopo la chiamata al secondo kernel si avrà la trasformata, per quel livello di iterazione, in memoria device (e quindi occorrerà ricopiare su memoria host). Banalmente, per una trasformata di Haar a k livelli, occorreranno $2k$ trasferimenti tra la memoria host e la memoria device.

Si noti inoltre che ciascun kernel, prende in input le dimensioni dell'immagine, mentre la griglia CUDA è allocata sempre della stessa dimensione (indipendentemente dal size del sottoquadrante dell'immagine su cui stiamo lavorando).

In ciascun kernel quindi vengono fatti lavorare soltanto i thread con id $<$ mxn, dove m ed n sono le dimensioni attuali dell'immagine da trasformare.

3.5 Trasformata di Haar dell'immagine in parallelo, senza trasferimenti multipli tra le memorie

Vogliamo ora modificare l'approccio utilizzato precedentemente, per implementare una trasformata di Haar, che utilizzi il calcolo parallelo in CUDA, limitando però quello che è un collo di bottiglia importante delle performance: il trasferimento di dati tra la memoria host e device.
Mentre l'approccio visto prima funziona bene per una trasformata di Haar 2D singola, è oneroso ed inefficiente quando si vuole iterare la trasformazione, in quanto ciascun livello della trasformata, necessità della copia in memoria device del sottoquadrante da trasformare e della copia sulla memoria host del risultato. Sebbene questo sia oneroso, in termini computazionali, fa sì che si possa lavorare con strutture dati lineari (gli array monodimensionali che rappresentano le immagini da trasformare) che hanno uno stride delle colonne che dipende dalla dimensione dell'immagine su cui stiamo lavorando.
Nel passare invece ad un'implementazione che calcoli tutti i livelli richiesti di trasformata, utilizzando sempre la stessa area di memoria in memoria device, occorrerà separare le dimensioni del sottoquadrante su cui stiamo lavorando (che rappresenta l'immagine da trasformare) dal numero effettivo di colonne per il quale tale immagine è allocata come vettore lineare.
Iniziamo modificando quelle che sono le regole per il calcolo degli indici, e proseguiamo spiegandone il significato, cercando di illustrarne la dinamica tramite un'immagine.
Fatto questo, le differenze rimanenti riguarderanno l'implementazione in C, e verranno illustrate nel seguito.
Innanzitutto quindi il calcolo degli indici $\overline{I}$ e $\overline{J}$ di un blocco 2x2 di pixel,a partire dall'indice monodimensionale del blocco logico, dovrà tenere conto del numero di colonne originali ($realN$) dell'immagine copiata (solo all'inizio) in memoria device, poichè questi indici rappresentano riga e colonna, del quadrato 2x2 di pixel, nell'immagine, indipendentemente che stiamo lavorando sull'immagine intera, o su un sottoquadrante di questa.

$$\overline{I} = \frac{idQuad}{(realN/2)},\ \bar{J} = idQuad\%\frac{realN}{2}$$

e allo stesso modo, il pixel che un thread, all'interno del kernel di ridisposizione (per la successiva lettura coalescente), deve spostare, va prelevato (dall'immagine da trasformare in input) all'indice:

$$(2 * \bar{I} + iSottogruppo) * realN + 2 * \bar{J} + jSottogruppo$$

Tale pixel viene posizionato in memoria globale all'indice (di una struttura dati fissa, grande quanto il vettore monodimensionale dell'immagine originale) uguale all'id globale del thread nella griglia

CUDA.
Una cosa importante è che in questo kernel (al quale vanno passati come parametri di input oltre che le dimensioni del sottoquadrante su cui stiamo lavorando, anche le dimensioni effettive delle colonne) il controllo per i thread che dovranno lavorare, verrà fatto non più utilizzando l'id globale del thread (nella griglia CUDA) rispetto al prodotto mxn delle dimensioni del sottoquadrante utilizzato (come avveniva nei kernel della prima versione), ma utilizzando gli indici $\overline{I}$ e $\overline{J}$, rispetto alle dimensioni di tale quadrante.Il motivo verrà chiarito tramite l'esempio grafico.
A questo punto parte il kernel che si occupa della trasformata di Haar.
Innanzitutto anche in questo kernel, come in quello della ridisposizione, il controllo dei thread che proseguiranno nell'elaborazione, viene fatto paragonando gli indici del blocco di 2x2 pixel assegnati, $\overline{I}$ e $\overline{J}$, rispetto alle dimensioni del quadrante da trasformare (m ed n, anche in questo caso ricevuti come parametri di input).
Una volta copiato il pixel assegnato al thread, nella memoria condivisa, ed aver calcolato il valore del pixel di output (così come veniva fatto per la prima versione del kernel) tale pixel di output verrà disposto in posizione:

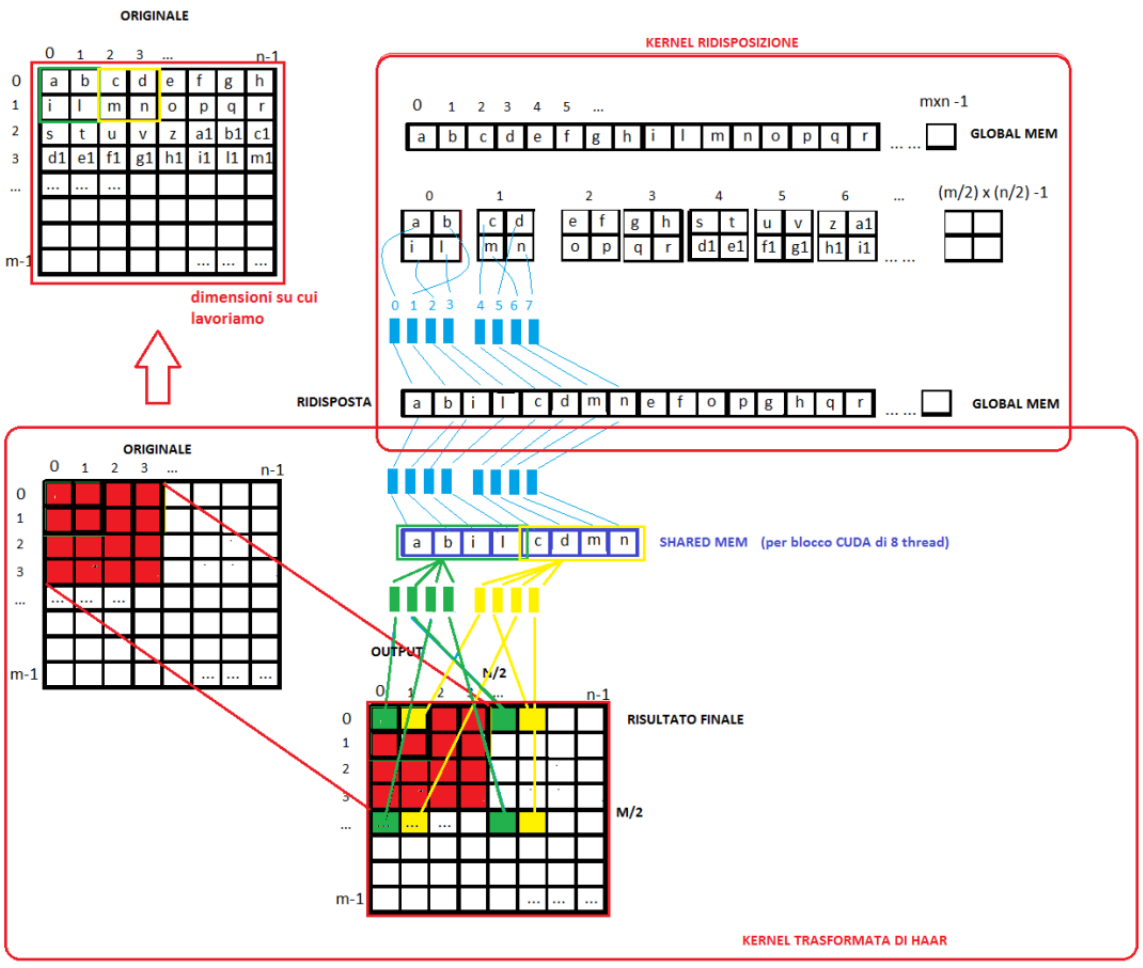
$$\bar{I} * realN + iSottogruppo * \frac{m}{2} * realN + \bar{J} + jSottogruppo * \frac{n}{2}$$

della struttura (grande quanto l'immagine originale, indipendentemente dal sottoquadrante su cui stiamo lavorando) che conterrà il risultato finale. Si noti come per lo stride nell'accesso si usi il numero reale di colonne, mentre per calcolare l'offset che indica in quale dei 4 sottoquadranti di output il pixel vada posizionato si usi il numero di colonne del sottoquadrante che stiamo trasformando al livello attuale della trasformata.
A questo punto, partendo da un'immagine di input, abbiamo ridisposto i pixel (col primo kernel) e li abbiamo trasformati, ottenendo l'immagine di output. Vogliamo iterare il procedimento, e per far ciò, copiamo il quadrante in alto a sinistra nella struttura dati (sempre quella originale grande quanto tutta la prima immagine di input).
Questa copia può essere fatta solo dai thread che risultano avere, indici riga e colonna all'interno del gruppo logico di 2x2 thread, pari a 0, e più precisamente il pixel ottenuto dall'ultima trasformazione, verrà posizionato all'indice

$$\bar{I} * realN + \bar{J}$$

della struttura che diventerà, alla prossima iterazione, l'immagine da trasformare.
Al passo successivo quindi, verranno dimezzate le dimensioni del sottoquadrante da trasformare (m

ed n) e verranno richiamati i 2 kernel sopra esposti.



Come notiamo nell'esempio grafico di sopra, si parte con l'immagine originale, relativamente alla quale lavoriamo con tutte le colonne (e righe).

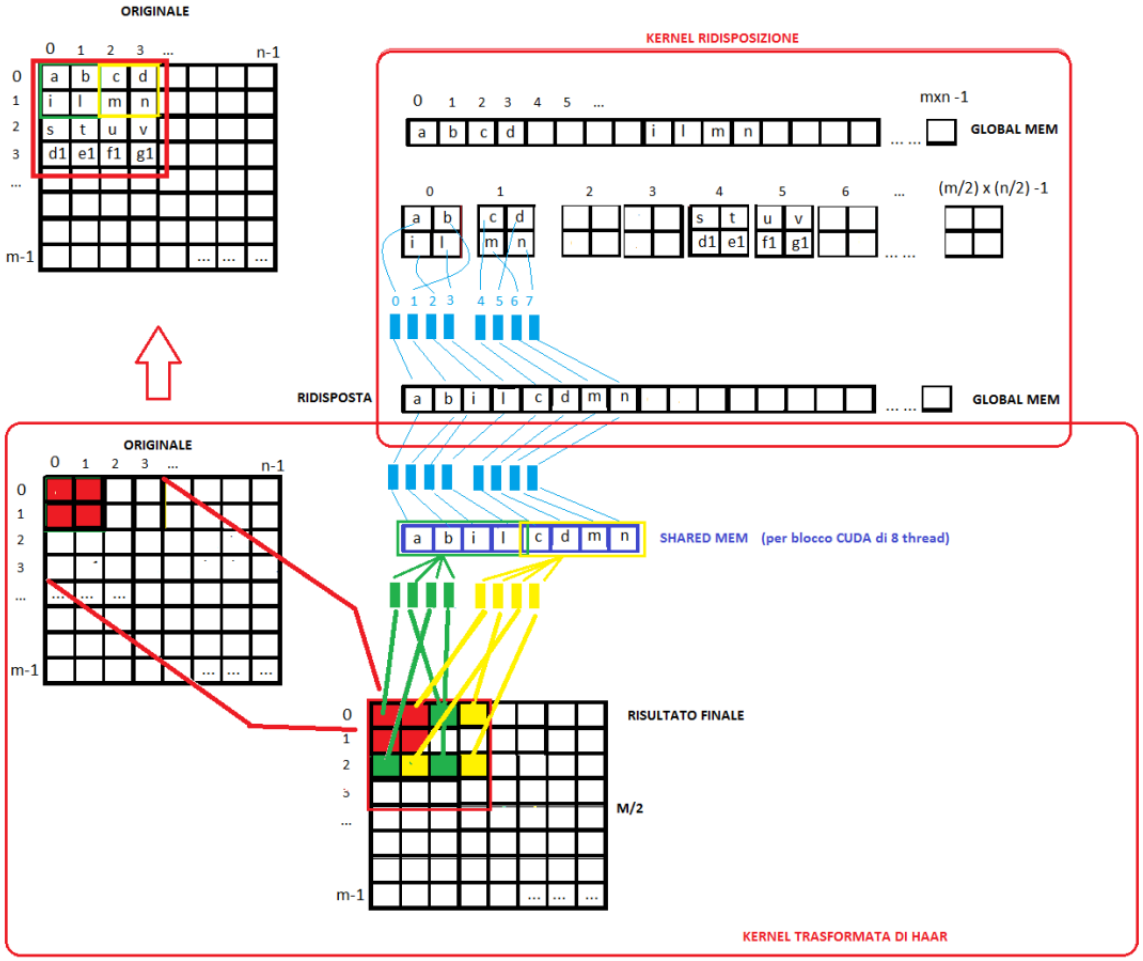
Passiamo dalla disposizione lineare (allocazione per righe) classica alla disposizione lineare che ci permetterà di leggere in maniera coalescente nel kernel di trasformazione (il kernel di ridisposizione gode di coalescenza nella fase di scrittura).

La ridisposizione in questione è effettuata quindi dal primo kernel.

Per semplicità visiva viene mostrata la disposizione lineare dei blocchi 2x2 di pixel che formano

l'immagine (tale disposizione non è presente in memoria per come mostrata, non essendocene bisogno), che permette di capire il mapping tra i thread, i blocchi, ed i pixel che ciascun thread (nel primo kernel) preleva e ridispone.
Alla fine del kernel di ridisposizione avremo l'array lineare dell'immagine ridisposto, in memoria globale su device.
Nel secondo kernel, ciascun thread copia il pixel assegnatogli nel blocco shared memory all'indice associato al suo id (id del blocco CUDA).
A questo punto un thread calcola il valore del pixel di output, utilizzando anche gli altri 3 pixel associati al suo blocco logico di lavoro, composto di 4 thread (gli altri 3 pixels saranno a questo punto disponibili in memoria shared).
Ciascun thread piazza il pixel di output in una porzione della struttura dati che conterrà il risultato finale (nell'immagine di sopra il risultato finale è rappresentato per semplicità come matrice 2D, ma si tratta pur sempre di un array lineare che la rappresenta, tramite uno stride fisso che è quello dell'immagine iniziale di partenza, anche per successivi livelli di iterazione della trasformata su sottoporzioni di tale immagine).
In ultima istanza, prima di terminare, ciascun thread del kernel di trasformazione che abbia posizionato il pixel di output nel primo quadrante in alto a sinistra, copierà tale valore anche nella struttura dati dell'immagine originale, che sarà l'input per la successiva iterazione, che ovviamente terrà conto delle nuove dimensioni della sottoimmagine da elaborare.

Il livello successivo della trasformata di Haar è illustrato nell'immagine seguente:



Si noti che dimezzando le dimensioni dell'immagine da elaborare, si nota la necessità di far lavorare soltanto i thread appartenenti a blocchi 2x2 contenuti in tale porzioni (in entrambi i kernel) pur rimanendo con uno stride di colonne costante (poichè le strutture dati con cui si lavora sono sempre le stesse).

4 Descrizione del software

Descriveremo in questo paragrafo l'implementazione adottata per l'applicazione, soffermandoci sugli snippet di codice più interessanti e sulle routine prodotte.

4.1 OpenCV

L'applicativo fa uso del framework OpenCV (c++) per la gestione delle matrici rappresentanti le immagini.
OpenCV permette di accedere facilmente ai valori dei pixel di un'immagine, fornendo direttamente una rappresentazione sia bidimensionale sia monodimensionale della matrice che la compone.
Un'immagine in OpenCV è rappresentata tramite un oggetto di classe cv::Mat , che rappresenta un header, una sorta di puntatore incapsulato, a quella che è la struttura dati sottostante, e che è possibile interrogare per ottenere informazioni relative al numero di righe e di colonne dell'immagine.
Nonostante l'ampia gamma di routine e strumenti nativi offerti da OpenCV, nel nostro caso ci serviremo delle sole routine di accesso ai pixel e di visualizzazione di un'immagine.
Prima di proseguire è bene sottolineare quali siano i tipi di immagini che tale framework può gestire, e qual è l'uso che ne faremo.
In OpenCV è possibile aprire (leggere da file system) un'immagine sia come immagine a colori, sia come immagine d'intensità. Nel primo caso si tratta di un'immagine associata ad una matrice contenente per ciascun pixel l'informazioni d'intensità nei 3 canali RGB, mentre nel secondo caso a ciascun pixel è associato un unico livello di intensità, e quindi si tratterà di un'immagine grayscale. Quest'ultimo tipo di immagine è quello che utilizzeremo, effettuando la trasformata di Haar per immagini grayscale (anche se il medesimo meccanismo si potrebbe estendere separatamente a ciascuno dei 3 canali).
Una volta scelto il tipo di immagine, occorre scegliere il tipo di dato che rappresenta il pixel. Per immagini grayscale il default è l'unsigned char, e questo sta a significare che per ogni pixel avremo una profondità di 8 bit (256 valori d'intensità possibili, tra 0 e 255). Tuttavia questo tipo di dato porta con sè un notevole errore di quantizzazione (la somma degli errori di arrotondamento) in quanto la nostra trasformata è più adatta all'elaborazione in virgola mobile.
Prima di iniziare l'elaborazione quindi, occorrerà convertire la cv::Mat in tipo float (da CV_8UC1, per l'unsigned char interpretato come un singolo canale di intensità, a CV_32FC1, ovvero 32 bit come singolo canale di intensità) in modo tale da ottenere non solo una maggiore precisione, ma una lettura coalescente nei kernel, essendo 1 byte una parola di memoria troppo piccola per soddisfare i requisiti di coalescenza[4]. Tuttavia sorge un problema nel momento in cui vorremo mostrare l'immagine a schermo, in quanto OpenCV visualizza correttamente immagini con pixel a valori in virgola mobile, soltanto se queste sono normalizzate tra 0 e 1 (mentre nel convertire l'immagine da CV_8UC1 a CV_32FC1 i valori rimangono tra 0 e 255). Per risolvere questo problema preve-

diamo una routine per la visualizzazione, che prende in input l'immagine (di tipo CV_32FC1) da visualizzare, e la riconverte in CV_8UC1.

4.2 Compilazione e parametri di input

Per compilare il sorgente (tutto contenuto nell'unico file main.cu) occorre linkare le librerie opencv. Si compila utilizzando nvcc in modo tale da ottenere un file eseguibile che sfrutti le librerie CUDA. Nel nostro caso, avendo lavorato da remoto ed in ambiente Linux, prima di compilare occorre collegarsi in ssh alla macchina.

ssh -X utente@redjeans.uniparthenope.it

ed in seguito, dopo essersi spostati nella cartella contenente il sorgente, si compila con (si faccia attenzione che ' è un backtick non un semplice apostrofo)

nvcc main.cu -o main 'pkg-config –libs opencv'

Per lanciare l'eseguibile (**./main**) occorre fornire 4 parametri di input, che corrispondono al

- **[path immagine]:** il path relativo, rispetto alla posizione dell'eseguibile, dell'immagine da trasformare
- **[livello]:** livello richiesto di nesting della trasformata di Haar. A seconda delle dimensioni dell'immagine, non è detto che sia possibile raggiungerlo in quanto l'applicativo si ferma nel momento in cui la sottoimmagine ottenuta non è divisibile s per 2.
- **[n. threads per blocco CUDA]:** il numero di threads per ciascun blocco CUDA. In totale avremo sempre un numero di threads pari al numero di pixel dell'immagine, divisi in tanti blocchi in base al valore scelto per questo parametro. Il valore di questo inoltre deve essere un numero maggiore o uguale di 4, e divisibile per 4. Si noti che altre limitazioni riguardano l'hardware CUDA, per le quali non è possibile avere un numero di blocchi maggiore di 65536 (in una dimensione, che è proprio il nostro limite superiore avendo pensato l'applicativo affinchè lavori con una griglia CUDA monodimensionale). Quindi si faccia attenzione che numeri di thread per blocchi CUDA troppo bassi, portano ad un numero di blocchi che potrebbe andare oltre il limite consentito.
- **[flag visualizzazione immagine]:** se si passa 0 le immagini non verranno visualizzate (nonostante le chiamate nel codice alla routine di visualizzazione). Questo accorgimento è

stato adottato per i casi di test con immagini estremamente grandi che potevano dare dei problemi in fase di visualizzazione, avendo testato l'applicativo da remoto. Passando 1 si effettuano le visualizzazioni (ove chiamate nel codice).

Indipendentemente dalle dimensioni delle righe e colonne dell'immagien scelta, per garantire che venga effettuato almeno un passo della trasformata di Haar, si riscala l'immagine alle più vicine dimensioni divisibili per 2 (nel caso in cui queste non lo siano). Tuttavia per successivi livelli di nesting non vengono effettuate operazioni di scaling, nel caso in cui le sottoporzioni non siano divisibili per 2, per evitare di perdere la simmetria con i precedenti livelli già calcolati (necessaria per effettuare la ricostruzione a partire dalla trasformata finale).

4.3 Routine implementate

4.3.1 Routine generali

Iniziamo a descrivere i metodi e le routines prodotte che costituiscono l'applicativo presentato. Iniziamo con le funzioni generiche, utilizzate sia nell'approccio sequenziale che in quello parallelo.

```
__host__ void mostraImmagine(const Mat & img, char *text);
__host__ void checkErroreCuda(char *msg);
```

La funzione **mostraImmagine** prende in input un alias di un oggetto di classe cv::mat, ne crea uno nuovo localmente (di tipo CV_8UC1) e vi copia (a seguito di conversione) l'immagine in input, visualizzandola e assegnando come titolo della finestra la stringa ricevuta in input come secondo parametro. *La visualizzazione dell'immagine potrebbe richiedere un po' di tempo in caso di traffico di rete intenso, ed inoltre, qualora si volesse chiudere la finestra e proseguire con l'elaborazione (la visualizzazione in opencv è bloccante di default) occorrerà selezionare la finestra e premere un tasto qualunque della tastiera.*

La seconda funzione **checkErroreCuda** prende in input il messaggio da stampare a video, e interroga il CUDA environment per ottenere un valore di tipo cudaError_t relativo all'ultimo errore verificatosi. Nel caso in cui non ci sia stato errore, la routine CUDA cudaGetLastError() ritorna cudaSuccess. In caso contrario stampiamo la stringa associata al valore cudaError_t ottenuto, che descriverà quindi l'ultimo errore verificatosi. Chiamando tale funzione dopo ogni chiamata ad una routine del framework CUDA, potremo eventualmente ottenere informazioni relative a situazioni di fault ed errori.

```
class MioTimer{
    cudaEvent_t startEvent,stopEvent;
    bool started;

    public:
        MioTimer();

        void start();

        float stopECalcolaTempo();

        ~MioTimer();

};
```

La classe sopra mostrata funge da wrapper per quello che è l'utilizzo previsto da CUDA per gli eventi..

Lanciare la routine CUDa cudaEventRecord fa si che venga inserito, nel workflow CUDA, un oggetto di tipo cudaEvent_t. Registrando più eventi di questo tipo, l'ordine a questi assegnato è del tipo FIFO. In tal modo l'ordine di registrazione degli eventi garantisce quale sarà l'ordine di evasione di questi. In tal modo è possibile chiamare la funzione CUDA cudaEventElapsedTime che restituisce il numero di ms trascorsi tra l'evasione dei 2 eventi nel workflow. Questo è il metodo utilizzato per valutare il tempo trascorso nell'elaborazione di un kernel o di qualunque altra operazione. Incapsulando tali primitive CUDA in una classe che permetta di avviare il conteggio del tempo (effettuando la registrazione del primo cudaEvent_t) e di stopparlo (registrando il secondo cudaEvent_t) abbiamo creato effettivamente una sorta di cronometro. Il metodo stopECalcolaTempo() stoppa quindi il timer e ritorna il tempo trascorso tra lo start e lo stop del timer. Si noti che è necessario sincronizzare il flusso dell'applicazione con la registrazione del cudaEvent_t di stop poichè chiamare la routine cudaEventRecord non assicura quale sarà il momento dell'effettiva evasione dell'evento passatogli.

4.3.2 Routines trasformata di Haar sequenziale

```
__host__ void haarTransformHost(const Mat &img,Mat outImg,float coeff)
```

Lo snippet sopra presenta una delle 2 versioni della routine che esegue la trasformata di Haar su host (cpu, in sequenziale)..

La routine prende in input l'oggetto cv::Mat da trasformare ed effettua la modifica su un'oggetto Mat ricevuto anch'esso in input.

Questa routine, così come l'altra versione che effettua sempre la trasformata in sequenziale, lavora

sull'immagine accedendo alla struttura sottostante come vera e propria matrice 2D.
Scorriamo quindi su tutti i pixel dell'immagine di output, e utilizzando i valori degli indici riga e colonna, calcoliamo quello che è il quadrante (sull'immagine di input) di 4 pixel contenente gli elementi da utilizzare (assieme ai coefficienti, a seconda della posizione del pixel di output) in combinazione lineare per ottenere il valore del pixel di output. Si noti che per ciascun pixel dell'immagine di output, è eseguito un controllo per vedere in quale dei 4 sottoquadranti ci troviamo e l'output corrisponde ad una convoluzione spaziale con un kernel 2x2.
Nel sorgente esiste (ed è effettivamente utilizzata) un'altra versione per il calcolo della trasformata in locale, più vicina a quella che è la schematizzazione fornita all'inizio della trasformata di Haar 2D: tale routine alternativa esegue prima una trasformazione di Haar 1D sulle colonne dell'immagine, e poi sulle righe del risultato precedentemente ottenuto (corrispondendo quindi ad una convoluzione con kernel 2x1).

La routine sopra mostrata esegue un solo livello della trasformata di Haar. Quindi abbiamo bisogno di una routine che iteri il processo di chiamata sui sottoquadranti contenenti solo average part (in alto a sinistra) fin quando si ottengano sotto porzioni di dimensioni divisibili per 2. Tale routine è mostrata di seguito:

```
__host__ void callRecursiveHaarHost(const Mat img,Mat finalRes,int & lvl,float coeff)
```

In questa funzione si utilizza un puntatore a cv::Mat iniziale che contiene una copia dell'immagine da trasformare. Iterando le chiamate alla routine che esegue la trasforma di Haar in sequenziale utilizzando una matrice di appoggio (quella puntata da res), si ricopia il risultato nella Mat finale, si estrae il quadrante in alto a sinistra e quello diventerà l'input per la successiva iterazione. Si prosegue fin quando si raggiunge il livello richiesto per il nesting della trasformata, oppure fin quando non sia possibile proseguire (aggiornando in tal caso lavariabile che indica il livello raggiunto). Si noti che la linea commentata, dove è presente la chiamata a mostraImmagine, permette di mostrare i risultati dei livelli di nesting intermedi della trasformata.

Abbiamo inoltre implementato una procedura (header sotto) che lavora su host (cpu), per ricostruire un'immagine a partire dalla trasformata di Haar di questa di un dato livello.
Non è stata fornita un'implementazione in CUDA per il processo di ricostruzione in quanto non richiesta la fase di ricostruzione dalla traccia del lavoro in esame (tuttavia abbiamo deciso di implementare ugualmente almeno una versione sequenziale per validare la bontà della trasformata effettuata).
Tale routine presenta la stessa struttura di quella per la trasformata di Haar 2D in sequenziale,

sia per la gestione della trasformazione inversa di un solo livello, che per una ricostruzione su più livelli.
La trasformata inversa relativa ad un solo livello è eseguita sfruttando anche in questo caso la separabilità del kernel 2D, e quindi lavorando prima sulle lungo le righe, e poi lungo le colonne (del risultato precedentemente ottenuto).

```
__host__ void reverseTransformHostV2(const Mat &img,Mat outImg,float coeff)
```

Relativamente alla routine che iterativamente richiama quella sopra mostrata, si rimanda all'equivalente routine per le chiamate iterate alla procedura per il calcolo della trasformata di Haar, avendo queste una struttura ed una logica simile.

4.3.3 Routines trasformata di Haar in parallelo, con copie intermedie

Illustriamo ora le procedura scritte per lavorare al problema in esame con CUDA, prevedendo trasferimenti multipli tra le memorie (device ed host) per ciascuno dei livelli di nesting della trasformata.

```
__global__ void ridisponiArray(float *imgDataOriginaleD,float *imgDataRidispostoD,int m,int n)
```

L'immagine di sopra presenta la routine utilizzata come kernel (verrà quindi eseguita sul device) per la ridisposizione dei pixel dell'immagine, in modo tale che l'effettivo kernel (illustrato sotto) per la trasformata di Haar possa effettuare una lettura coalescente dei pixel (secondo la logica spiegata all'inizio). Tale kernel prende in input l'array monodimensionale che rappresenta l'immagine da trasformare, l'array nel quale salvare la ridisposizione, e le dimensioni dell'immagine su cui stiamo lavorando. Si noti che essendo questa la versione che prevede il ritrasferimento, in memoria host, del risultato ottenuto, tale kernel prenderà in input, ad ogni livello di iterazione, un array monodimensionale allocato con uno stride pari proprio al numero di colonne della sottoimmagine su cui stiamo lavorando (diversamente da quanto avviene per i kernel che non prevedono trasferimenti multipli tra le memorie).

```
__global__ void trasformataHaarGPU(float * imgDataD,float * trasformataDataD,int m,int n,float coeff)
```

Qui abbiamo invece il kernel che si occupa dell'effettiva trasformata di Haar, prendendo in input l'array contenente i pixel ridisposti, l'array sul quale salvare la trasformata della sottoimmagine in input, e le dimensioni di tale immagine, oltre che il coefficiente da utilizzare per il banco di filtri di Haar. Anche in questo caso, trovandoci nel caso della trasformata in parallelo con trasferimenti tra

le memorie host e device per ogni livello di nesting, lo stride effettivo della struttura contenente i pixel dell'immagine, è pari proprio alle dimensioni per le quali stiamo lavorando.

```
__host__ Mat * callHaarDeviceOneStep(const Mat & img, const dim3 & sizeBlocco,
                                     const dim3 & sizeGriglia,float coeff)
```

Questa funzione invece è sempre relativa ad un singolo livello di nesting della trasformata di Haar, e si occupa di allocare la memoria sul device (e deallocarla) e di chiamare prima il kernel di ridisposizione dei pixel e poi quello di trasformazione. La funzione prende in input un'immagine (sottoforma di oggetto cv::Mat) che è proprio il sottoquadrante su cui stiamo lavorando al livello attuale di nesting, e ritorna puntatore alla cv::Mat del risultato. La routine inoltre prende in input i parametri relativi alla configurazione della griglia CUDA, decisi all'avvio dell'applicazione.

```
__host__ Mat * callRecursiveHaarDeviceCopieIntermedie (const Mat & img,int & lvl, const dim3 & sizeBlocco,
                                                       const dim3 & sizeGriglia, float coeff)
```

Ed infine abbiamo la routine che si occupa di chiamare iterativamente le funzioni precedentemente illustrate, in modo tale da ottenere un nesting delle trasformate di Haar per l'immagine di partenza. Tale funzione prende in input l'immagine originale di partenza, e altri parametri quali il coefficiente dei banchi di filtering e la configurazione della griglia CUDA, restituendo in output il puntatore alla cv::Mat del risultato finale. In maniera del tutto simile a come avveniva per la versione sequenziale, questa routine divide via via l'immagine, estraendo di volta in volta il sottoquadrante dell'ultima trasformata ottenuta, e passandolo come immagine di input per il successivo livello di nesting, alla routine callHaarDeviceOneStep. Si noti che all'interno del codice di questa routine viene fatto partire il conteggio dei tempi. E' inoltre possibile, decommentnado l'opportuna chiamata alla funzione mostraImmagine, ottenere una stampa del risultato al livello di nesting raggiunto (si faccia attenzione che in quel caso i tempi non saranno più coerenti, rimanendo il flusso d'esecuzione bloccato in attesa dell'input dell'utente da tastiera).

4.3.4 Routines trasformata di Haar in parallelo, senza copie intermedie

La logica delle routine che implementano la trasformata di Haar, evitando le copie intermedie, ma lavorando sempre con la stessa struttura dati (in memoria device) dall'inizio alla fine del processo iterativo di nesting, è stata già precedentemente illustrata. Le routine utilizzate si differenziano da quelle che utilizzano copie intermedie per il calcolo che si fa degli indici, che in alcuni punti hanno bisogno di conoscere l'effettivo stride, ovvero il numero di colonne reale delle strutture dati. Tale informazione quindi viene presa in input dai 2 kernel implementati. Le funzioni utilizzate sono le seguenti:

```
__global__ void ridisponiPorzione(float *imgDataDaRidisporre, float *imgDataRidispostoD, int m,int n,int realM,
                                  int realN);

__global__ void trasformataHaarGPUv2(float *imgDataRidispostoD,float * trasformataDataD,float *nextQuad,int m,
                                     int n,int realM,int realN,float coeff);

__host__ Mat * callHaarDeviceAllSteps(const Mat & img,int & lvl,float coeff,dim3 sizeGriglia, dim3 sizeBlocco);
```

Il principale collo di bottiglia, che nella versione con copie intermedie avveniva nella routine callHaarDeviceOneStep, che si occupava di allocare la memoria su device per ogni livello di nesting, e di copiare e ricopiare i dati da e verso le memoria dell'host, viene evitato allocando un'unica volta la memoria device (ed effettuando quindi un'unica copia verso la memoria gpu ed un'unica copia verso la memoria host) nella routine callHaarDeviceAllSteps, che si occupa inoltre di eseguire la suddivisione delle dimensioni del quadrante di lavoro relativo al livello di nesting attuale.

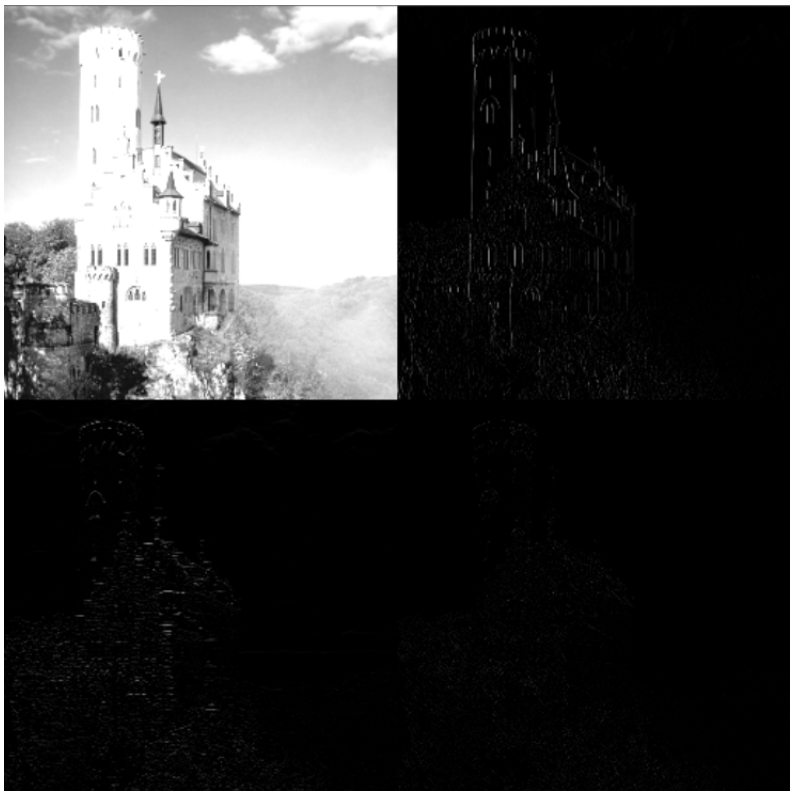
5 Test, esempi e calcolo dei tempi

Illustriamo di seguito una serie di esempi di esecuzione, ed i tempi (misurati in *millisecondi*) calcolati utilizzando la classe MioTimer appositamente scritta.
Per ciascuna immagine, vengono misurati i tempi per il calcolo della trasformata di Haar sia CPU (**Host**), sia su GPU (sia con copie multiple tra le memorie, **CUDA v1**, sia senza copie multiple, **CUDA v2**) facendo variare il numero di thread per ciascun blocco CUDA.

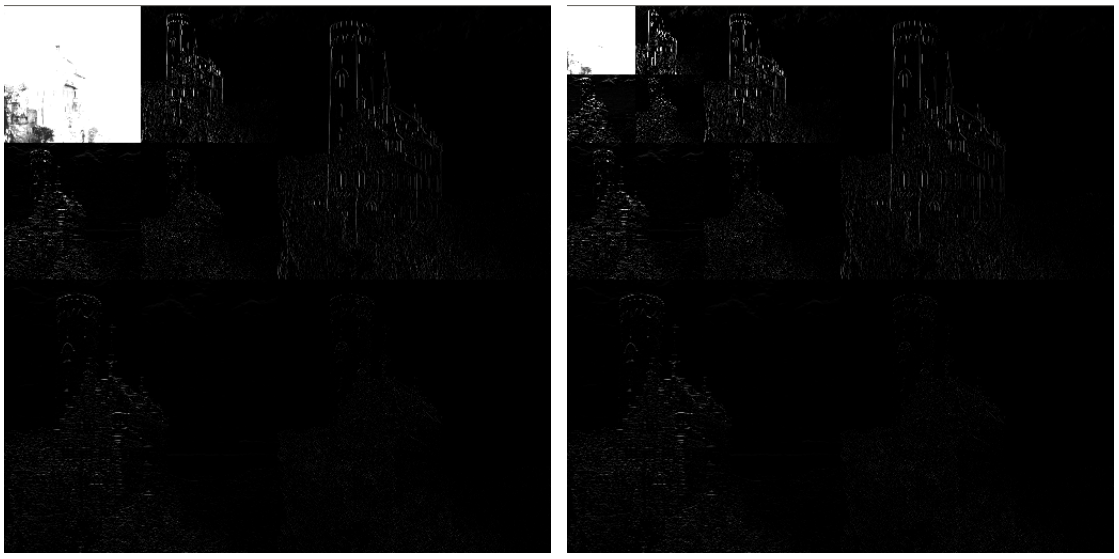
5.1 Test

Di seguito viene mostrata una carrellata di immagini, ciascuna seguita dalle trasformate di Haar di un dato livello.

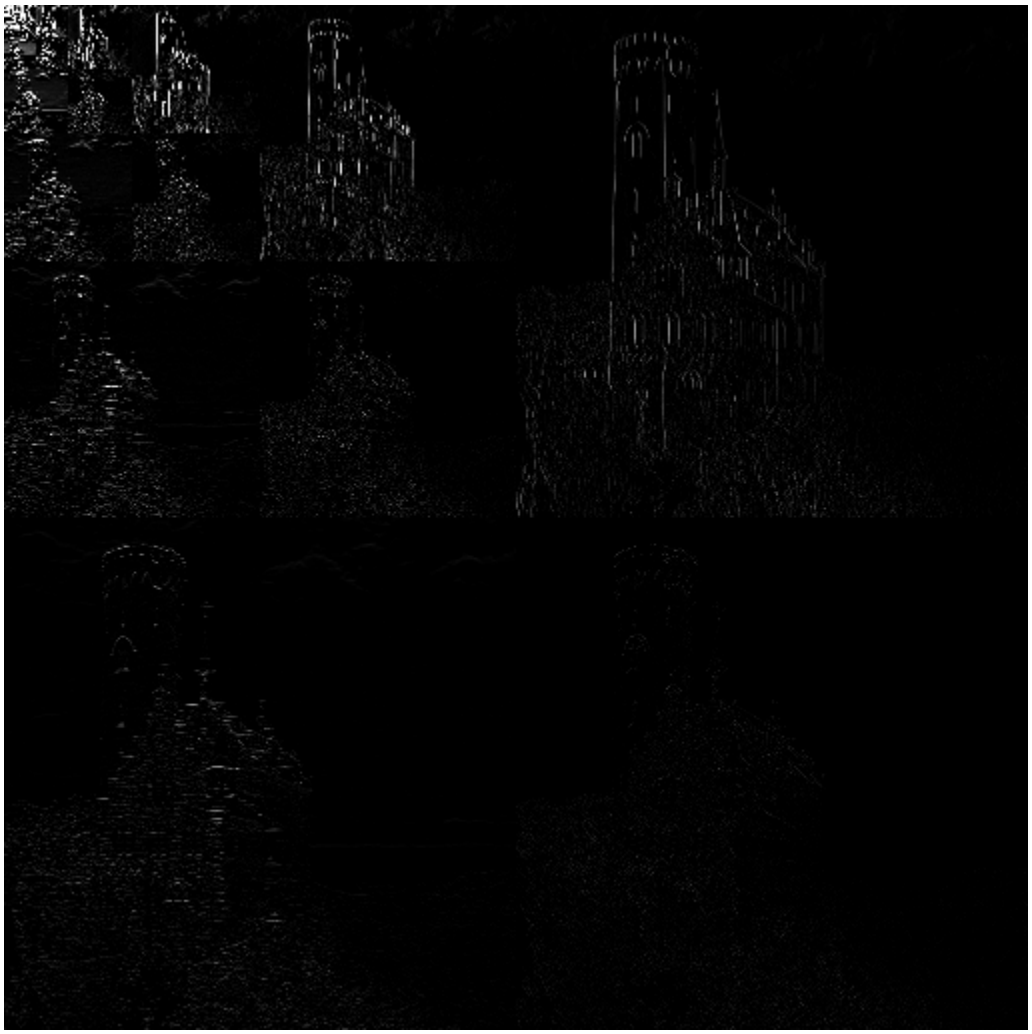
Immagine originale (img.png)



Trasformata di Haar di primo livello di img.png



Trasformate per i 2 livelli successivi di img.png

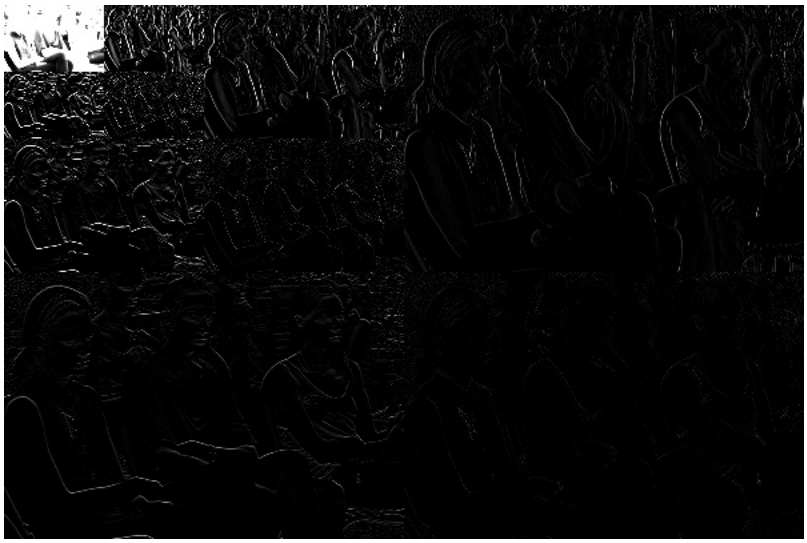


Trasformata di livello 9 di img.png

Immagine originale (img3.png)

trasformata di primo livello di img3.png

Immagine originale img5.jpg



Trasformata di livello 3 di img5.jpg

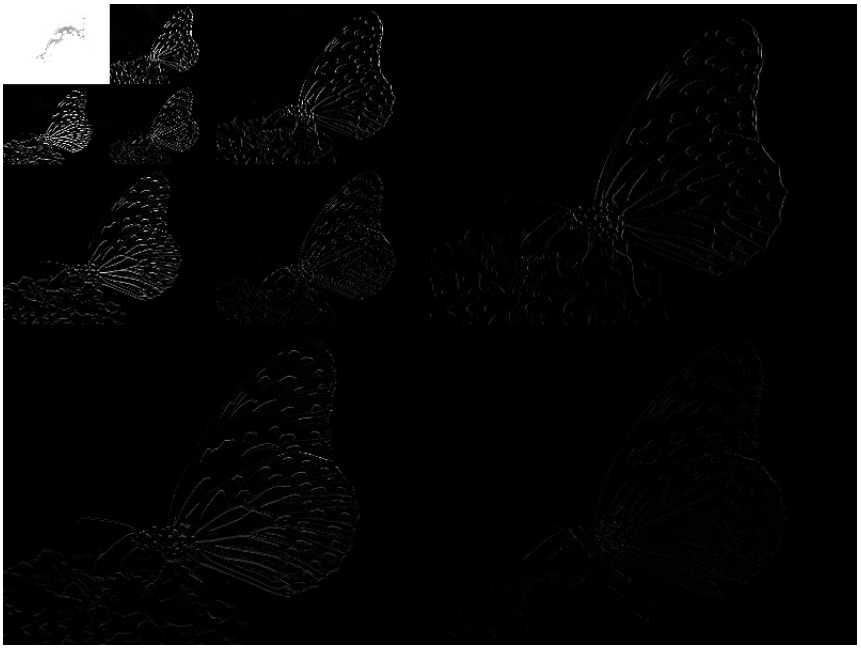
Immagine originale lena.jpg

Trasformata di livello 8 di lena.jpg

Immagine originale debbie.jpg

Trasformata di livello 8 di debbie.jpg

Immagine originale img4.jpg



Trasformata di img4.jpg (la parte bassa dell'immagine è tagliata poichè sulla macchina di test non era possibile visualizzare interamente l'immagine)

5.2 Calcolo dei tempi

Immagine	Livello	Host	CUDA v1 8 threads	CUDA v2 8 threads	CUDA v1 32 threads	CUDA v2 32 threads	CUDA v1 128 threads	CUDA v2 128 threads	CUDA v1 256 threads	CUDA v2 256 threads
img3.png **250x256**	1	4.22	1.66	0.54	1.06	0.31	1.72	0.30	1.78	0.29
debbie.jpg **256x256**	8	5.19	4.72	1.51	4.85	0.88	4.02	0.46	3.65	0.48
foto1.jpg **512x512**	9	19.08	12.55	4.72	9.12	3.04	8.75	2.35	8.51	1.70
img.png **512x512**	9	21.60	13.17	5.75	10.02	2.42	8.57	2.05	8.46	1.46
img5.jpg **400x600**	3	21.22	7.91	3.04	6.89	1.54	7.19	1.29	7.25	0.93
flower.jpg **400x620**	3	23.22	8.05	3.22	7.41	1.65	8.32	1.41	7.84	0.92
img4.jpg **600x800**	3	43.65	15.32	5.08	14.02	2.17	14.31	1.66	12.46	1.33
wall3.jpg **1700x1000**	2	140.16	-	-	41.99	4.77	40.14	3.30	40.22	3.27
wall.jpg **1800x2880**	3	458.34	-	-	-	-	107.56	10.22	108.25	10.75

Qui abbiamo una tabella riassuntiva che mostra i tempi di calcolo delle trasformate di Haar per i massimi livelli di nesting ottenibili per determinate immagini, al variare del numero di threads per blocco CUDA.

I tempi sono calcolati in millisecondi, e sono comprensivi dei tempi richiesti per il trasferimento tra le memorie host e device nelle versioni che sfruttano CUDA.

Le colonne relative a **CUDA v1** sono quelle che presentano i tempi per la versione CUDA con copie multiple per ciascun livello di nesting, tra le memorie host e device, mentre **CUDA v2** è relativo alla versione senza copie intermedie.

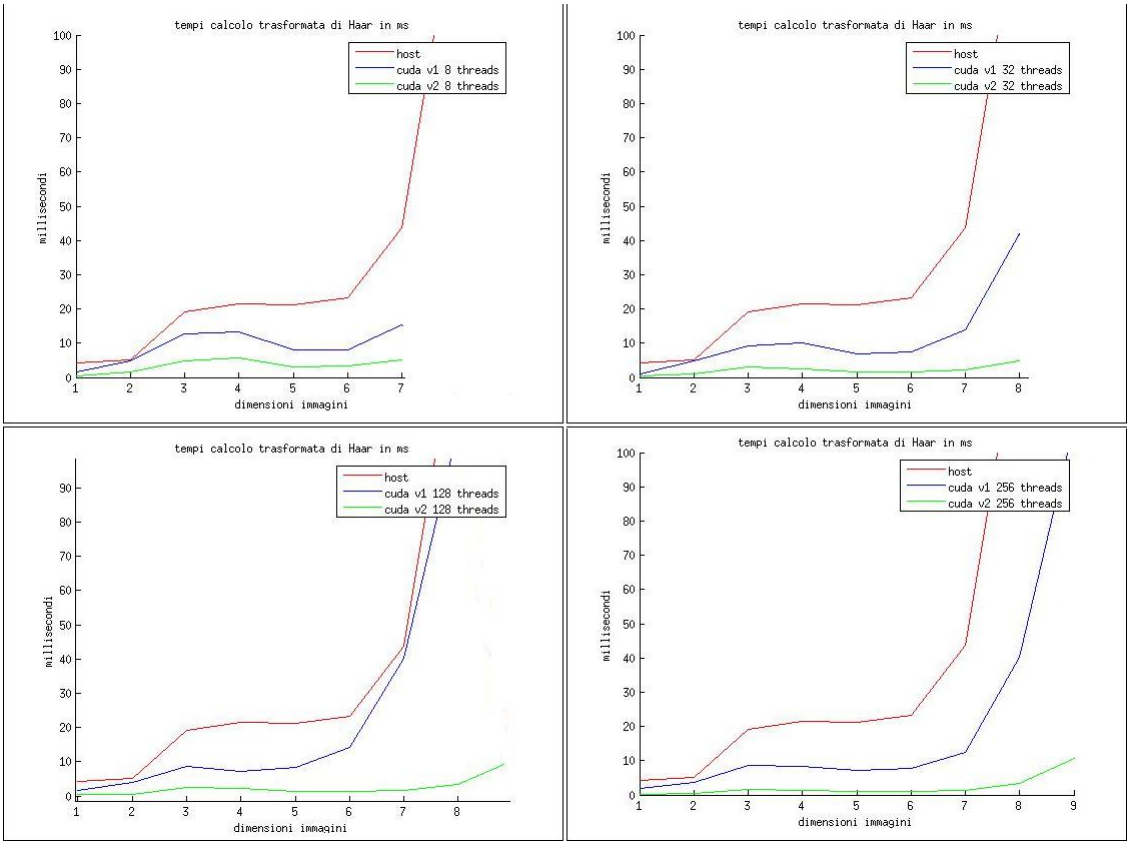
La colonna **Host** presenta i tempi di calcolo della versione sequenziale su CPU.

Appare evidente, osservando i tempi (indipendentemente dalla configurazione della griglia CUDA) l'incredibile guadagno prestazionale ottenuto passando dalla versione su CPU, a quella su GPU, utilizzando la seconda versione, quella con u unico passaggio tra le memorie host e device iniziale, ed un unico finale (indipendentemente dal livello di nesting richiesto). Si passa ad esempio da 4 ms circa per un'immagine di 256x256 per la versione sequenziale, a 0.3 ms circa per la versione parallelizzata (efficientemente).

Relativamente alla versione con copie multiple tra device e host, risulta evidente l'enorme peso in termini prestazionali della fase di copia tra le memorie host e device, e della gestione dei dati lato host, che in tale versione è ripetuto per ciascun livello di nesting.

Infatti si nota che anche quando si ha un unico livello di nesting, la versione senza copie intermedie risulta più veloce. Questo è dovuto al fatto che nella versione senza copie intermedie, i thread appartenenti a ciascun kernel copiano direttamente i pixel di output nella posizione della struttura dati finale, mentre relativamente alla versione con copie intermedie, lavorando i kernel su strutture dati specifiche della sottoimmagine relativa ad un dato livello di nesting, sarà necessario ricopiare (su host) gli output intermedi sulla struttura dell'immagine finale..

Di seguito sono mostrati 4 grafici dei tempi (per ciascuna delle 3 versioni) al variare del numero di processori. I grafici presentano il plotting dei tempi (in ms) rispetto al size delle immagini presentate nello stesso ordine della tabella(relativamente a ciascuna immagine va tenuto in considerazione il livello di nesting massimo calcolato, come riportato sempre in tabella).



Il trend rilevabile è che per immagini con un solo (o pochi) livelli di nesting richiesti, la versione CUDA con copie intermedie offre un buon guadagno rispetto alla versione sequenziale (ad esempio img3.png), in quanto il vantaggio ottenuto nel calcolo in parallelo, non viene perso nella fase di passaggio dei dati tra le memorie (in quanto si ha un livello di nesting ridotto). All'aumentare del livello delle dimensioni dell'immagine, pur aumentando leggermente il livello di nesting, viene

mantenuto tale guadagno in quanto la perdita prestazionale dovuta al leggero aumento del livello di nesting, è bilanciata da una perdita molto maggiore per la versione host (in quanto all'aumento lineare di una singola dimensione, il numero di pixel aumenta in maniera quadratica, e questo incide soltanto sulla versione host, mentre per la versione CUDA con copie multiple la ridisposizione è fatta sempre in parallel).

Il massimo collo di bottiglia, per la versione CUDA con trasferimenti multipli, lo si ha quindi nel caso di immagini piccole, con un elevato livello di nesting richiesto. L'elevato livello di nesting richiede innumerevoli operazioni di passaggio dati tra le memorie host e device (completamente assenti nella versione su CPU) ed un elevatissimo overhead (lato host) nell'interpretazione dei risultati parziali forniti dai kernel di quel livello.

E' possibile valure l'effettivo numero di chiamate ripetute alla routine CUDA di trasferimento di dati tra le memorie, nella versione con copie multiple, utilizzando lo strumento **nvprof** del CUDA framework.

Tale strumento permette di effettuare un profiling di quello che è l'utilizzo dell'hardware GPU. .

Mostriamo qui di seguito il profiling dell'applicazione ottenuto lanciando il profiling dell'applicativo..

Nella prima immagine abbiamo il profiling prima della versione CUDA con copie multiple, e poi quello senza copie multiple, per l'immagine debbie.jpg, con un solo livello di nesting richiesto.

```
C129@beowulf:~/testprog$ nvprof ./main debbie.jpg 1 128 0
======== NVPROF is profiling main...
======== Command: main debbie.jpg 1 128 0
calcolo trasformata per immagine 256 x 256
tempo: livello trasformata: 1 usando GPU , numero threads per blocco: 128 con cudaMemcpy intermed
======== Profiling result:
 Time(%)      Time   Calls       Avg       Min       Max  Name
   34.88   46.69us       1   46.69us   46.69us   46.69us  [CUDA memcpy DtoH]
   33.90   45.38us       1   45.38us   45.38us   45.38us  [CUDA memcpy HtoD]
   16.40   21.95us       1   21.95us   21.95us   21.95us  trasformataHaarGPU(float*, float*, int,
    9.87   13.22us       1   13.22us   13.22us   13.22us  ridisponiArray(float*, float*, int, int
    4.95    6.62us       1    6.62us    6.62us    6.62us  [CUDA memset]
C129@beowulf:~/testprog$ nvprof ./main2 debbie.jpg 1 128 0
======== NVPROF is profiling main2...
======== Command: main2 debbie.jpg 1 128 0
calcolo trasformata per immagine 256 x 256
tempo: livello trasformata: 1 usando GPU , numero threads per blocco: 128 senza copie intermedie
======== Profiling result:
 Time(%)      Time   Calls       Avg       Min       Max  Name
   30.75   45.60us       1   45.60us   45.60us   45.60us  [CUDA memcpy HtoD]
   30.38   45.06us       1   45.06us   45.06us   45.06us  [CUDA memcpy DtoH]
   15.75   23.36us       1   23.36us   23.36us   23.36us  trasformataHaarGPUv2(float*, float*, fl
t)
   12.45   18.46us       3    6.15us    5.70us    6.82us  [CUDA memset]
   10.66   15.81us       1   15.81us   15.81us   15.81us  ridisponiPorzione(float*, float*, int,
C129@beowulf:~/testprog$
```

E qui l'immagine del profiling per l'elaborazione della stessa immagine, con 8 livelli di nesting. Si noti il numero di chiamate alle routine CUDA memcpy HtoD e DtoH che è pari al numero di livelli,

nella prima chiamata all'applicativo.

```
C129@beowulf:~/testprog$ nvprof ./main debbie.jpg 8 128 0
======== NVPROF is profiling main...
======== Command: main debbie.jpg 8 128 0
calcolo trasformata per immagine 256 x 256
tempo: livello trasformata: 8 usando GPU , numero threads per blocco: 128 con cudaMemcpy intermed
======== Profiling result:
 Time(%)      Time   Calls       Avg       Min       Max  Name
   26.04   77.15us       8    9.64us    2.34us   46.05us  [CUDA memcpy DtoH]
   23.22   68.80us       8    8.60us    1.34us   45.66us  [CUDA memcpy HtoD]
   21.78   64.54us       8    8.07us    5.25us   21.82us  trasformataHaarGPU(float*, float*, int,
   17.20   50.98us       8    6.37us    4.96us   13.44us  ridisponiArray(float*, float*, int, int
   11.75   34.82us       8    4.35us    3.87us    6.66us  [CUDA memset]
C129@beowulf:~/testprog$ nvprof ./main2 debbie.jpg 8 128 0
======== NVPROF is profiling main2...
======== Command: main2 debbie.jpg 8 128 0
calcolo trasformata per immagine 256 x 256
tempo: livello trasformata: 8 usando GPU , numero threads per blocco: 128 senza copie intermedie
======== Profiling result:
 Time(%)      Time   Calls       Avg       Min       Max  Name
   34.34  109.60us       8   13.70us   11.94us   23.07us  trasformataHaarGPUv2(float*, float*, fl
t)
   31.60  100.83us       8   12.60us   11.90us   15.81us  ridisponiPorzione(float*, float*, int,
   14.22   45.38us       1   45.38us   45.38us   45.38us  [CUDA memcpy HtoD]
   14.08   44.93us       1   44.93us   44.93us   44.93us  [CUDA memcpy DtoH]
    5.77   18.40us       3    6.13us    5.70us    6.82us  [CUDA memset]
C129@beowulf:~/testprog$
```

5.3 Considerazioni

Quello che in definitiva se ne deduce, è che, come d'altronde sarebbe intuitivo aspettarsi già leggendo la documentazione ufficiale CUDA, è che conviene effettuare il maggior numero di operazioni possibili nei kernel, cercando di ridurre al minimo le chiamate a questi, oltre che le operazioni di copia tra le memorie.

Inoltre non basta ridurre al minimo tali trasferimenti affinchè un applicativo sfrutti pienamente la potenza del calcolo parallelo con CUDA, ma occorre anche una progettazione mirata per l'algoritmo. Infatti, guardando ad esempio nella prima versione del nostro kernel, all'overhead dovuto ai trasferimenti multipli tra le memorie, va ad aggiungersi il costo computazionale della riorganizzazione (lato host) del risultato (parziale, relativo a quel livello) della trasformazione relativa all'ultima immagine, poichè in tale versione è stato previsto che i kernel lavorino su strutture dati create ad hoc, ad ogni livello, prendendo in input array monodimensionali con uno stride pari al numero di colonne su cui lavorare. Tale approccio quindi risulta non completamente efficiente, poichè richiede un'elaborazione lato host che renda trasparente al kernel il lavoro tra più livelli, ma questo appunto fa si che le prestazioni degradino considerevolmente.

Infine si nota anche come l'applicativo sia poco dipendente dal numero di thread scelti per blocco CUDA (o non sia dipendente quanto ci si aspetterebbe in generale da un applicazione CUDA) in

quanto il tipo di problema in analisi è intrinsecamente adatto ad essere scomposto in *blocchi logici* di lavoro da 4 thread ciascuno.

Relativamente alle possibili modifiche e/o sviluppi futuri, si potrebbe pensare di implementare un allocazione che faccia uso del *pitch,* per valutare eventuali cambiamenti nelle prestazioni. Tuttavia tale modifica sarebbe facilmente implementabile in quanto già abbiamo a disposizione kernel (quelli della versione ottimizzata) che lavorano con strutture dati caratterizzate da uno stride diverso dal numero di colonne rappresentanti l'immagine in analisi.

6 Bibliografia

[1] Wavelets, Signal Processing, Fourier Transforms, George Dallas, UK based Information Engineering

[2] Haar Function, WolframMathWorld

[3] Image Compression: How Math Led to the JPEG2000 Standard, Haar Wavelet Transformation, whydomath.org

[4] The Haar-Wavelet Transform in Digitale Image Processing: Its Status and Achievements, Piotr Porwik, Agnieszka Lisowka Institute of Informatics, University of Silesia