Equalizzazione dell'istogramma di un'immagine a colori: versione sequenziale e versione parallela

Alessandro Sestini
Matricola - 6226094
alessandro.sestini@stud.unifi.it

Abstract

Questo elaborato si concentra sullo studio e sull'implementazione di una strategia per rendere un'immagine visivamente piú omogenea equalizzando l'istogramma dei valori dei pixel. L'implementazione é stata fatta sia in Java che in C/CUDA e in entrambi i casi sono state fatte le versioni sequenziali e parallele. Al termine delle implementazioni sono state effettuate delle analisi delle performance di tutte le versioni: per la parte in Java, il programma é stato testato su una macchina con processore Intel i5 a 2 core, mentre per la parte in CUDA é stato provato su una GPU NVDIA Titan X con 12 Gb di RAM e 3072 CUDA cores. Infine é stato valutato lo speedup ottenuto passando dalla versione sequenziale alla versione parallela in entrambi i linguaggi.

1. Introduzione

Il miglioramento di un'immagine é un problema che risulta essere molto importante per il mondo dell'image processing: l'obiettivo principale é quello di migliorare visivamente l'immagine aumentando il contrasto globale delle figure. Un metodo per ottenere questo risultato é l'Equalizzazione dell'Istogramma: supponendo di avere un'immagine a scala di grigi, si basa sostanzialmente sul calcolare l'istogramma dei valori dei pixel, rendere uniforme questo grafico e mappare i vecchi livelli di grigio in nuovi valori dettati dall'istogramma equalizzato stesso.

1.1. Istogramma ed equalizzazione

L'istogramma di un'immagine a livelli di grigi compresi tra [0, 255] é una funzione discreta $h(v_k) = n_k$ per $k = 0, .., 255$, dove v_k é un livello di grigio e n_k é il numero dei pixel che assumono quel valore. Intuitivamente é ragionevole pensare che un'immagine che tende a occupare tutto il range dei possibili valori di grigio e che tende ad essere distribuita uniformemente, avrá un alto contrasto e mostrerá una grande varietá di tonalitá di grigi: sará quindi visivamente migliore. La funzione che rende l'istogramma uniforme é definita come:

$$h_{eq}(v) = round(\frac{cdf(v)-cdf(0)}{(M*N)-1} * L - 1)$$

dove $cdf(v)$ é la funzione di distribuzione cumulativa dell'istogramma originale, $M * N$ é la grandezza dell'immagine e $L-1$ é il massimo valore che puó assumere un pixel (tipicamente 255). Una volta fatta l'equalizzazione, si mappano i vecchi pixel nei nuovi valori:

$$v_{new} = h_{eq}(v_{old})$$

Nella Fig. 1 é possibile vedere il risultato:

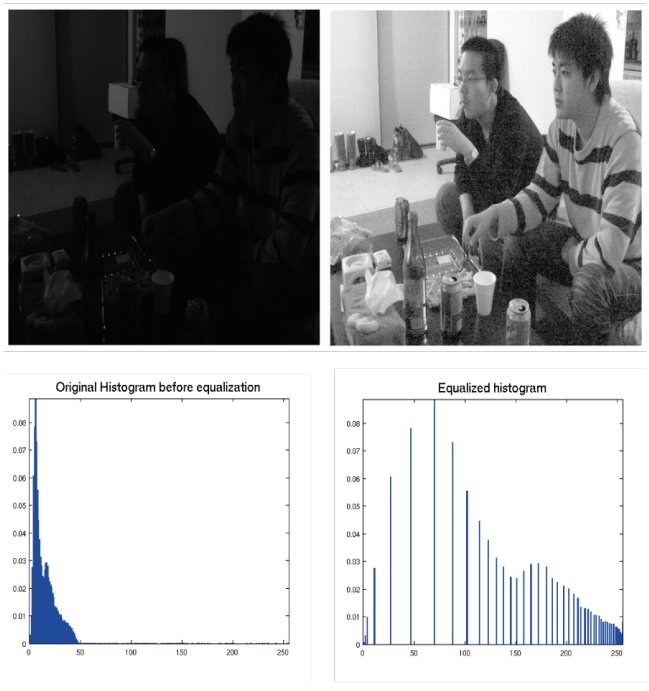

Fig. 1. Risultato dell'equalizzazione.

1.2. Equalizzazione di un'immagine a colori

Fin'ora abbiamo parlato solamente di immagini a scale di grigi, ma é possibile applicare lo stesso procedimento a immagini a colori. Supponiamo di avere una figura a 3 canali RGB: per effettuare un'equalizzazione é necessario prima fare una conversione dello spazio colore e passare allo spazio YCbCr. La conversione per valori compresi tra [0, 255] é definita come:

$$\begin{cases} Y = 0.299 * R + 0.587 * G + 0.114 * B \\ Cb = 128 - 0.168736 * R - 0.331264 * G + 0.5 * B \\ Cr = 128 + 0.5 * R - 0.418688 * G - 0.081312 * B \end{cases}$$

dove Y rappresenta la componente di luminanza e Cb e Cr le componenti di crominanza. Per rendere l'immagine piú omogena basta dunque equalizzare il canale Y, lasciando inalterate le componenti di colore dell'immagine. Una volta trovati i nuovi valori di luminanza equalizzati, basta tornare indietro nello spazio RGB ottenendo cosí un'immagine a colori migliore. Le equazioni che legano i valori RGB a YCbCr (sempre compresi tra [0, 255]) sono:

$$\begin{cases} R = Y + 1.402 * (Cr - 128) \\ G = Y - 0.344136 * (Cb - 128) - 0.714136 * (Cr - 128) \\ B = Y + 1.772 * (Cb - 128) \end{cases}$$

In Fig. 2 é possibile vedere il risultato del processo:

Fig. 2. Equalizzazione immagine a colori.

2. Implementazione

Passiamo ora all'implementazione del processo: analizzeremo prima la versione sequenziale, ed essendo molto simile per entrambi i linguaggi verrá esaminata una volta sola, poi passeremo alla versione parallela prima in Java e in seguito in CUDA.

2.1. Versione sequenziale

La versione sequenziale dell'algoritmo é piuttosto semplice e intuitiva: viene caricata un'immagine da file; per ogni pixel dell'immagine si fa la conversione da RGB a YCbCr mentre si costruisce l'istogramma del canale Y; una volta ottenuto, viene fatta l'equalizzazione dell'istogramma e, per ogni pixel dell'immagine, viene mappato il nuovo valore di luminanza secondo il nuovo istogramma per poi fare la conversione da YCbCr a RGB. In Java l'immagine caricata viene trattata come un oggetto *BufferedImage*, da cui si estraggono le righe per questioni di performance, mentre in C é trattato come un array monodimensionale di *char*. Di seguito é riportato lo pseudocodice dell'algoritmo:

```
im = Load('file.jpg');
width = im.getWidth();
height = im.getHeight();

for x = 0:width-1
  for y = 0:height-1
    (R, G, B) = im(x,y);

    Y = 0.299 * R + 0.587 * G + 0.114 * B;
    Cb = 128 - 0.168736 * R - 0.331264 * G + 0.5 * B;
    Cr = 128 + 0.5 * R - 0.418688 * G - 0.081312 * B;

    im(x,y) = (Y,Cb,Cr);

    histogram[Y] ++;
  end
end

sum = 0;
for i = 0:255
  sum += histogram[i];
  hist_equalized[i] = (sum - histogram[0])/
                      (width*height - 1) * 255;
end

for x = 0:width-1
  for y = 0:height-1
    Y = hist_equalized(im(x,y).Y);
    Cb = im(x,y).Cb;
    Cr = im(x,y).Cr;

    R = Y + 1.402 * (Cr - 128);
    G = Y - 0.344 * (Cb - 128) - 0.714136 * (Cr - 128);
    B = Y + 1.772 * (Cb - 12));

    im(x,y) = (R,G,B);

  end
end
```

2.2. Considerazioni

I due cicli che prevedono la conversione dell'immagine e il calcolo dell'istogramma sono le operazioni che richiedono piú calcoli e piú tempo, e quindi l'obiettivo é in sostanza parallelizzare questi elementi. Il for di mezzo serve ad equalizzare l'istogramma ed é praticamente un ciclo su un array di 256 elementi, troppo pochi per poter pensare di avere un miglioramento evidente, ma é comunque un buon esercizio di parallelizzazione. La Tab. 1 mostra il tempo medio trascorso dal programma sequenziale all'interno dei cicli per un'immagine di grandi dimensioni (7680x4320):

	Tempo
Ciclo 1	0.458 s
Ciclo 2	0.001 s
Ciclo 3	0.379 s

Tab. 1. Tempo medio trascorso dal programma all'interno dei cicli.

3. Versione parallela - Java

L'obiettivo é parallelizzare i 3 cicli, e dunque l'idea in Java é semplicemente quella di usare 3 diverse tipologie di thread.

3.1. Primo thread

La prima tipologia provvede alla conversione dell'immagine dallo spazio RGB allo spazio YCbCr mentre calcola l'istogramma del canale Y. Ogni thread prende una parte dell'immagine (Fig. 3), la converte e restituisce un istogramma calcolato privatamente della propria sezione. Per fare questo é neccesario che i thread implementino l'interfaccia *Callable<Int[]>*, di modo da poter restituire un array di int, che verrá poi recuperato nella parte sequenziale da un oggetto di tipo *Future<int[]>*; il risultato di ogni Callable sará poi sommato per ottenere l'istogramma completo dell'immagine. L'esecuzione dei thread viene gestita tramite un Thread Pool di dimensione fissa uguale al numero di processori disponibile nella macchina.

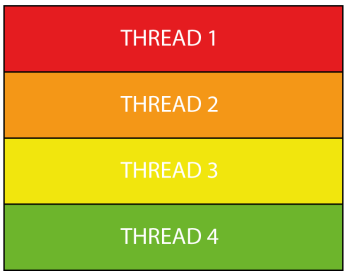

Fig. 3. Suddivisione dell'immagine da parte dei thread. Ogni thread processerá una parte indipendente, per cui non sono possibili race condition; se il numero dei thread non é multiplo dell'altezza dell'immagine, l'ultimo thread lavorerá su una parte piú piccola

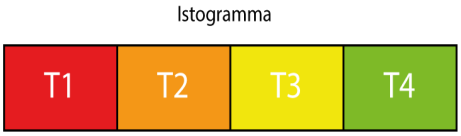

Fig. 4. Suddivisione dell'istogramma da parte dei thread della seconda tipologia.

Di seguito viene riportato il codice Java della prima tipologia:

```
public int[] call(BufferedImage sub_image) {

  //In ingresso avremo solamente una parte dell'immagine
  int width = sub_image.getWidth();
  int height = sub_image.getHeight();
  for (int y = 0; y < height; y++) {
    //Si estrae una riga di pixel
    // e si itera su questa
    sub_image.getRaster().getPixels(0,y,width,1,iarray);
    for (int x = 0; x < iarray.length; x+=3) {
      //Si effettua la conversione e
      //si calcola l'istogramma privato
      int r = iarray[x+0];
      int g = iarray[x+1];
      int b = iarray[x+2];

      int Y = (int)(0.299*r+0.587*g+0.114*b);
      int Cb = (int)(128-0.168736*r-0.331264*g+0.5*b);
      int Cr = (int)(128+0.5*r-0.418688*g-0.081312*b);

      iarray[x+0] = Y;
      iarray[x+1] = Cb;
      iarray[x+2] = Cr;

      this.histogram[Y]++;
    }
    sub_image.getRaster().setPixels(0,y,width,1,iarray);
  }
  return histogram;
}
```

3.2. Secondo thread

La seconda tipologia equalizza l'istogramma: anche in questo caso viene usato un Thread Pool di dimensione fissa (multipla di 256) in cui ogni thread prende in ingresso la funzione di distribuzione cumulativa, precalcolata nella parte sequenziale, l'istogramma e lavorerá su una porzione dell'istogramma equalizzato (inizializzato a 0). Cosí come la tipologia precedente, ogni thread processerá la propria parte, e quindi le operazioni fatte sono totalmente indipendenti tra di loro rendendo cosí impossibili race condition. In Fig.4 é possibile vedere la suddivisone dell'istogramma equalizzato da parte dei thread.

3.3. Terzo thread

L'ultima tipologia é essenzialmente uguale alla prima: ogni thread lavora su una parte dell'immagine mappando i vecchi valori del canale Y nei nuovi dettati dall'istogramma equalizzato, per poi effetture la conversione da YCbCr a RGB. Come nei casi precedenti, viene usato un Thread Pool di dimensioni fisse uguale al numero dei processori disponibili.

3.4. Analisi delle performance

Un elemento da tenere conto nella parallelizzazione in Java é la creazione dei thread: questa richiede delle risorse e delle operazioni che nella versione sequenziale non abbiamo, causando cosí dei rallentamenti nella parte parallela.

Per questo per immagini di piccole dimensioni il guadagno non si nota, ma quando si processa immagini molto grandi (si parla di immagini 8k), il vantaggio é sostanziale, ottenendo uno speedup di circa 2 su una macchina a 2 core. Nella Tab. 2 sono riassunti i risultati medi degli esperimenti (i tempi sono presi senza considerare il caricamento dell'immagine ma fanno riferimento solamente al tempo di equalizzazione):

	Sequenziale	Parallelo
800x600	0.059 s	0.080 s
1200x1600	0.150 s	0.125 s
3840x2160	0.340s	0.250 s
7680x4320	0.950 s	0.530 s

Tab. 2. Risultati medi degli esperimenti in Java.

4. Versione parallela - CUDA

Anche in CUDA l'obiettivo principale é quello di parallelizzare i 3 cicli for, ma in maniera completamente diversa da quanto fatto in Java. Si usano infatti 3 diversi kernel (ricordiamo che in C l'immagine viene salvata come un buffer di char):

4.1. Primo kernel

Prima di parlare dei kernel, dobbiamo pensare a gestire la memoria della GPU: si allocano quindi un array di char per l'immagine in input e 3 array di int di lunghezza 256 per salvare rispettivamente l'istogramma dell'immagine, la funzione di distribuzione cumulativa e l'istogramma equalizzato; si provvede poi con *cudaMemCpy* a copiare l'immagine dall'host al device. In seguito si avvia il primo kernel: come primo passo si crea un array nella memoria shared che viene condiviso dai thread all'interno di un blocco per calcolare privatamente l'istogramma e si inizializza i valori di questo a 0, per poi chiamare un *synctrheads()*. A questo punto si converte l'immagine nello spazio colore desiderato. Se il numero di thread é uguale o maggiore al numero dei pixel, ogni thread processerá un solo valore. Altrimenti se il numero dei thread é piú piccolo delle dimensioni dell'immagine, ogni thread processerá sezioni contigue, per poi passare tutti insieme alla sezione successiva (Fig. 5). Di seguito l'estratto del codice:

```
__shared__ int hist_priv[256];
... Inizializzazione hist_priv ...

__synchtreads();

//Indice del thread
int idx = blockIdx.x*blockDim.x + threadIdx.x;
//Numero totale dei thread
int stride = blockDim.x*gridDim.x;
for(int i = idx; i < width*height; i += stride){
  int point_index = i*3;
  int r = input[point_index];
  int g = input[point_index+1];
  int b = input[point_index+2];

  int Y = (int)(0.299*r+0.587*g+0.114*b);
  int Cb = (int)(128-0.168736*r-0.331264*g+0.5*b);
  int Cr = (int)(128+0.5*r-0.418688*g-0.081312*b);

  atomicAdd(&(hist_priv[Y]), 1);
}

__synchtreads();
```

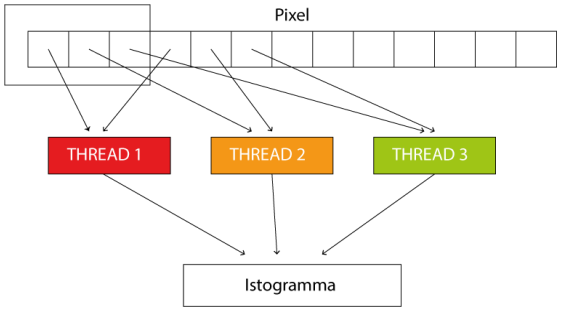

Fig. 5. Suddivisione dell'immagine da parte dei thread in CUDA.

Come ultimo passo si riunisce l'istogramma shared nella struttura dati originale nella memoria globale:

```
for(int bin_idx = threadIdx.x; bin_idx < 256;
        bin_idx += blockDim.x){
  atomicAdd(&(hist[bin_idx]), hist_priv[bin_idx]);
}
```

4.2. Accesso coalesced e istogramma shared

Vale la pena soffermarci a questo punto su due aspetti:

1. **Accesso coalesced**: come abbiamo detto, l'immagine viene salvata in un buffer di char, che viene poi salvato nella memoria globale. Uno dei fattori principali delle performance dei kernel é l'accesso a questo tipo di memoria, molto piú lente rispetto alla memoria shared. Le moderne DRAM usano un parallelismo per incrementare il rate d'accesso: ogni volta che una locazione viene acceduta, viene acceduto ugualmente un range

	Java		Cuda	
	Sequenziale	Parallelo	Sequenziale	Parallelo
800x600	0.059 s	0.080 s	0.018 s	0.002 s
1200x1600	0.150 s	0.125 s	0.080 s	0.007 s
3840x2160	0.340 s	0.250 s	0.321 s	0.025 s
7680x4320	0.950 s	0.530 s	1.100 s	0.100 s

Tab. 3. Tabella riassuntiva dei risultati degli esperimenti in entrambi i linguaggi.

di locazioni consecutive, chiamate burst, che include anche quella richiesta. Quindi, se i thread di un warp accedono a locazioni consecutive che sono nello stesso burst, allora verrá fatta una sola richiesta alla DRAM. Nel nostro caso, come é possibile vedere nella Fig. 5, l'accesso alla memoria é assicurato essere coalsced.

2. **Istogramma shared**: Le memorie cache delle GPU sono gli strumenti principali per ridurre la latenza degli accessi alla memoria globale, per questo viene calcolato l'istogramma all'interno della memoria shared che viene condivisa da tutti i thread di un blocco. Il problema peró é che gli aggiornamenti fatti dai thread non sono visibili all'interno di altri blocchi. L'idea della privatizzazione é far sí che ogni thread possa avere la propria copia privata a livello di blocco e possa effettuare delle operazioni atomiche su di essa; queste copie possono incrementare il throughput soprattutto per operazioni che necessitano di essere atomiche, ma il fatto negativo é che devono essere riunite alle strutture dati originali dopo che i calcoli sono stati completati.

4.3. Secondo e terzo kernel

Una volta che il primo kernel é terminato, si copia l'istogramma nella memoria globale dal device all'host, in CPU si calcola la funzione di distribuzione cumulativa e si passa questo nuovo array dall'host al device. A questo punto si avvia il secondo kernel: la prima operazione che viene fatta é semplicemente quella di applicare la funzione per equalizzare l'istogramma:

```
for(int i = idx; i < 256;
        i += blockDim.x*gridDim.x){
  hist[i] = (int) (((((float)cdf[i] - cdf[0]))/
        (((float)width*height - 1)))*255);
}
```

Una volta calcolato, si avvia il terzo kernel dove si mappano i vecchi valori del canale Y nei nuovi valori e si converte l'immagine da YCbCr a RGB. Il procedimento é sostanzialmente lo stesso del primo kernel, quindi non verrá riportato il codice.

4.4. Analisi delle performance

La piú grande differenza dalla versione Java é che in CUDA il tempo perso nella creazione dei thread é molto minore, quindi é possibile vedere un miglioramento nell'esecuzione anche per immagini di piccole dimensioni. Nella Tab. 4 vengono riportati i risultati medi degli esperimenti effettuati con una Titan X con 12 Gb di RAM e 3072 CUDA Cores:

	Sequenziale	Parallelo
800x600	0.018 s	0.002 s
1200x1600	0.080 s	0.007 s
3840x2160	0.321 s	0.025 s
7680x4320	1.100 s	0.100 s

Tab. 4. Risultati medi degli esperimenti in CUDA.

5. Conclusioni

In questo elaborato abbiamo dunque sperimentato il processo di equalizzazione dell'istogramma per immagini a colori parallelo, sia tramite i thread in Java sia tramite CUDA, trasformando l'immagine da RGB a YCbCr ed equalizzando l'istogramma dei valori del canale Y. Gli studi e gli esperimenti hanno dimostrato che in Java otteniamo uno speedup lineare di circa 2 su una macchina a 2 core solamente per immagini molto grandi a causa del tempo perso nella creazione dei thread, mentre in CUDA il guadagno temporale ottenuto tramite una Titan X é di circa 10 volte piú alto rispetto alla versione sequenziale anche per immagini di piccole dimensioni. Nella Tab. 3 sono riassunti i risultati degli esperimenti comparando le due implementazioni.

6. Note

Nell'implementazione Java non viene usata nessun libreria esterna per caricare le immagini, mentre in C viene usato OpenCV solamente per leggere l'immagine da file e salvarla in un array di char. Il buffer cosí creato sará composto da $width * height * 3$ valori, nella sequenza $pixel_1.R, pixel_1.G, pixel_1.B$, e cosí via. Alla fine dell'algoritmo verrá salvata l'immagine equalizzata.