Finalterm Parallel - Histogram Equalization

Edoardo Fanciullacci

April 2025

Introduzione

L'elaborazione digitale delle immagini rappresenta una disciplina chiave in numerosi ambiti applicativi, dalla fotografia digitale alla diagnostica per immagini, dalla videosorveglianza all'elaborazione di dati satellitari. Uno degli obiettivi principali in questo contesto è il miglioramento del contrasto visivo, ossia l'ampliamento della differenza percepibile tra le aree chiare e scure di un'immagine, al fine di facilitare l'interpretazione del contenuto visivo.

Una tecnica particolarmente efficace per il miglioramento del contrasto è l'**Histogram Equalization**, ovvero l'equalizzazione dell'istogramma. Tale tecnica modifica la distribuzione delle intensità dei pixel in modo da sfruttare uniformemente tutto l'intervallo disponibile (tipicamente da 0 a 255 per immagini a 8 bit), mettendo così in evidenza i dettagli nascosti in regioni sotto o sovraesposte.

Nel caso di immagini a colori, è prassi comune applicare l'equalizzazione sul canale di *luminanza* Y piuttosto che sui singoli canali RGB. Per questo motivo, si effettua una conversione dallo spazio dei colori RGB allo spazio **YUV**, separando il canale Y (luminanza) dalle componenti U e V (crominanza). In questo modo si migliora il contrasto percepito mantenendo la coerenza cromatica dell'immagine.

Il processo di equalizzazione si articola nelle seguenti fasi:

- **Conversione da RGB a YUV**: si isola la componente Y su cui verrà eseguita l'equalizzazione.
- **Calcolo dell'istogramma e della CDF**: si conta quanti pixel assumono ciascun valore di intensità (0–255) nel canale Y e si calcola la somma cumulativa (*Cumulative Distribution Function*, CDF).
- **Costruzione e applicazione della Look-Up Table (LUT)**: la CDF viene normalizzata per ottenere una funzione di mapping:

$$LUT[i] = \text{round}\left(\frac{CDF(i) - CDF_{\min}}{N - CDF_{\min}} \cdot 255\right)$$

dove $CDF_{\min}$ è il primo valore non nullo della CDF e N è il numero totale di pixel.

- **Conversione da YUV a RGB**: l'immagine equalizzata viene riconvertita in RGB per la visualizzazione finale.

In questo progetto, il processo di Histogram Equalization è stato implementato sia su **CPU** che su **GPU**, con l'obiettivo di confrontare le prestazioni e valutare l'impatto di diverse strategie di ottimizzazione, tra cui l'accesso coalescente alla memoria, l'uso della memoria condivisa e la gestione della parallelizzazione tramite CUDA.

1 Risultati Visivi e Istogrammi

Figure 1: Immagine originale (*Prima*) e immagine equalizzata (*Dopo*).

La trasformazione mostrata nella Figura 1 evidenzia in modo chiaro l'effetto dell'equalizzazione dell'istogramma. Nella prima immagine, quella originale, si osservano molte aree sottoesposte, con un'illuminazione non omogenea che impedisce di distinguere efficacemente i dettagli nelle zone d'ombra, come il tetto della struttura o le sedute in primo piano.

Dopo l'applicazione dell'equalizzazione, la scena risulta visivamente più bilanciata. Il canale di luminanza Y, isolato nello spazio colore YUV, è stato trasformato attraverso una funzione di mappatura basata sulla CDF (Cumulative Distribution Function). Tale trasformazione ha consentito una redistribuzione dei livelli di intensità, migliorando il contrasto e rendendo le informazioni visive più leggibili senza alterare le componenti cromatiche U e V.

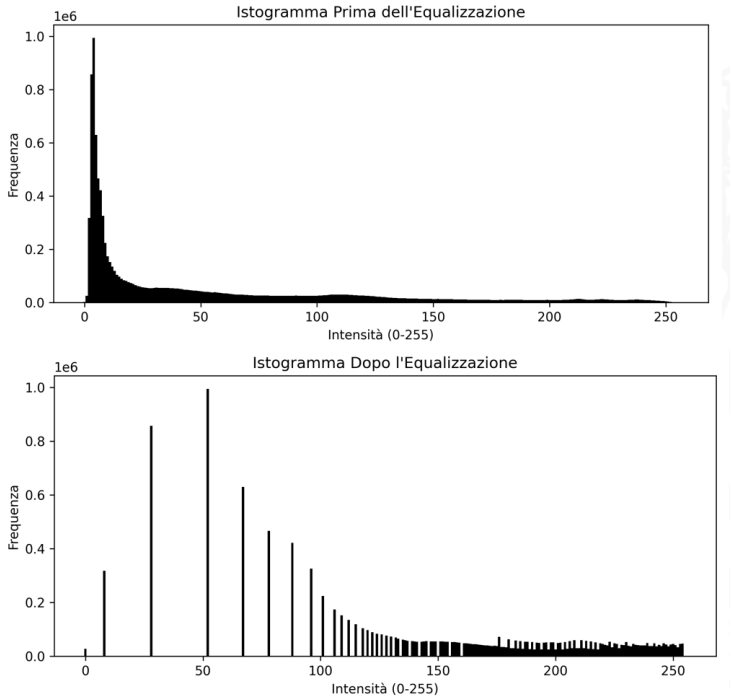

Figure 2: Istogramma del canale Y prima e dopo l'equalizzazione.

In Figura 2 è possibile osservare l'evoluzione dell'istogramma dell'immagine. Prima dell'equalizzazione, la distribuzione dei pixel è fortemente sbilanciata verso le intensità più basse: gran parte dei pixel presenta un livello di luminanza compreso tra 0 e 50, indicando una predominanza di toni scuri. Dopo l'equalizzazione, l'istogramma si espande sull'intera gamma (0–255), con una distribuzione più uniforme. Questo effetto, pur non producendo un istogramma perfettamente piatto (cosa teoricamente desiderabile), rappresenta un significativo aumento della gamma dinamica percepita.

In sintesi, l'algoritmo ha migliorato la qualità visiva dell'immagine, distribuendo meglio le informazioni di luminanza e aumentando la leggibilità delle zone precedentemente scure. Questo è particolarmente vantaggioso in immagini con illuminazione non uniforme o forti contrasti, migliorando l'esperienza visiva senza alterare i colori originari.

2 Analisi della Versione Sequenziale

La versione sequenziale del programma di equalizzazione dell'istogramma rappresenta il punto di partenza concettuale dell'implementazione. Essa è progettata per eseguire l'intero processo su CPU, seguendo un approccio monothreaded. Questo approccio si presta bene all'analisi del flusso logico dell'algoritmo ed è

ideale per comprendere le basi del funzionamento prima di introdurre tecniche di parallelizzazione.

2.1 Gestione dei dati

L'immagine viene caricata utilizzando la libreria `stb_image.h`, che restituisce un array lineare di byte, codificato nel formato RGB interleaved (`[R, G, B, R, G, B, ...]`). Questo layout viene gestito in modo esplicito nel ciclo di conversione RGB → YUV, dove ogni tripla di byte viene letta sequenzialmente per ciascun pixel. La separazione in tre vettori distinti (`Y`, `U`, `V`) permette di operare in modo indipendente sul solo canale di luminanza Y, come richiesto dall'algoritmo di equalizzazione.

L'immagine viene poi trasformata in tre array distinti `Y`, `U` e `V`, contenenti rispettivamente luminanza e crominanza.

2.2 Conversione RGB → YUV

La conversione avviene mediante una trasformazione lineare sui canali RGB:

```
const float yf = 0.299f * rf + 0.587f * gf + 0.114f * bf;
const float uf = -0.14713f * rf - 0.28886f * gf + 0.436f * bf + 128.0f;
const float vf = 0.615f * rf - 0.51499f * gf - 0.10001f * bf + 128.0f;
```

Figure 3: Conversione RGB-YUV

I valori vengono successivamente clampati nel range [0, 255] per garantire la validità nel dominio a 8 bit.

2.3 Costruzione dell'Istogramma

L'istogramma viene costruito con un semplice ciclo che conta la frequenza di ciascun valore Y:

```
std::vector histogram( n: 256, value: 0);
for (int i = 0; i < size; ++i)
    histogram[Y[i]]++;
```

Figure 4: Costruzione Istogramma

2.4 Calcolo della CDF e della LUT

La CDF viene calcolata tramite una prefix sum, mentre la LUT applica la normalizzazione:

```cpp
std::vector cdf(256, 0);
cdf[0] = histogram[0];
for (int i = 1; i < 256; ++i)
    cdf[i] = cdf[i-1] + histogram[i];

int cdf_min = 0;
for (int i = 0; i < 256; ++i) {
    if (cdf[i] != 0) {
        cdf_min = cdf[i];
        break;
    }
}
std::vector<unsigned char> lut(256);
for (int i = 0; i < 256; ++i) {
    float val =(cdf[i] - cdf_min) /(size - cdf_min) * 255.0f;
    lut[i] =(std::clamp(val, 0.0f, 255.0f));
}
```

Figure 5: Calcolo CDF e LUT

2.5 Equalizzazione e Conversione Inversa

Applicazione della LUT al canale Y e riconversione in RGB:

```cpp
for (int i = 0; i < size; ++i)
    Y[i] = lut[Y[i]];
```

Figure 6: Equalizzazione e YUV-RGB

2.6 Osservazioni sul Parallelismo

La versione sequenziale non prevede parallelismo: ogni operazione viene eseguita in modo deterministico su un singolo core. Ciò rende il codice semplice e comprensibile, ma penalizza le prestazioni su immagini ad alta risoluzione, dove il tempo di elaborazione può diventare significativo.

2.7 Output e Visualizzazione

L'immagine finale viene salvata in formato JPEG. Inoltre, gli istogrammi prima e dopo l'equalizzazione vengono esportati su file e rappresentati tramite uno script Python per facilitarne l'analisi visuale.

Listing 1: Esecuzione script di plotting

```
system("../venv/bin/python3 ../plot_histograms.py");
```

3 Analisi della Versione `CUDA Basic`

La prima implementazione parallela dell'algoritmo di equalizzazione dell'istogramma sfrutta le capacità della GPU attraverso l'uso delle API CUDA, mantenendo però una struttura semplice e diretta. La funzione host `run_gpu_version_basic` gestisce tutte le operazioni necessarie, suddividendole in kernel CUDA distinti per ciascuna fase del processo. Questa versione costituisce il punto di partenza per l'ottimizzazione successiva e permette una comprensione chiara delle problematiche legate al parallelismo su GPU.

3.1 Allocazione e gestione della memoria

I dati dell'immagine sono caricati sulla CPU e trasferiti alla GPU con un layout lineare interleaved (`unsigned char*`), dove ciascun pixel è rappresentato da tre byte consecutivi (R, G, B). La memoria sulla GPU viene allocata tramite `cudaMalloc` e l'immagine `input` è copiata dalla RAM del computer nella memoria GPU tramite `cudaMemcpy`, come nel seguente frammento:

```
cudaMalloc(&d_inputRGB, size: size * 3 * sizeof(unsigned char));
cudaMemcpy(d_inputRGB, input, count: size * 3 * sizeof(unsigned char), kind: cudaMemcpyHostToDevice);
```

La separazione delle componenti YUV viene realizzata allocando tre array lineari distinti (`d_Y`, `d_U`, `d_V`), con accesso indipendente per ciascun canale.

3.2 Conversione RGB → YUV

Il kernel `convertRGBtoYUV` (`__device__`) assegna a ogni thread il compito di elaborare un pixel. Viene eseguita una trasformazione lineare del colore, utilizzando coefficienti precalcolati caricati in memoria costante (`__constant__`). Tuttavia, l'accesso al buffer `d_inputRGB` non è coalescente, poiché i thread consecutivi di un warp accedono a byte distanziati (3 per pixel):

```
const unsigned int globalIdx = idx * 3;
const unsigned char r = input[globalIdx + 0];
const unsigned char g = input[globalIdx + 1];
const unsigned char b = input[globalIdx + 2];
```

Questo layout produce un elevato numero di transazioni di memoria non coalescenti, penalizzando le performance complessive.

3.3 Calcolo dell'istogramma

Il kernel `computeHistogram` calcola la distribuzione delle intensità del canale Y sfruttando la memoria condivisa. Ogni blocco alloca un array condiviso di 256 interi (`localHisto`) inizializzato da tutti i thread del blocco:

```
__shared__ unsigned int localHisto[NUM_BINS];
const unsigned int threadId = threadIdx.x;

if (threadId < NUM_BINS) {
    localHisto[threadId] = 0;
}
__syncthreads();
```

L'aggiornamento dell'istogramma è effettuato in modo concorrente tramite `atomicAdd`, evitando conflitti su memoria globale ma non su quella condivisa. In output, l'istogramma è scritto su memoria globale:

```
atomicAdd(&localHisto[Y[idx]], 1);
...
atomicAdd(&histo[threadId], localHisto[threadId]);
```

3.4 Calcolo CDF e costruzione della LUT

Dopo il calcolo dell'istogramma, questo viene copiato sulla CPU tramite `cudaMemcpy`. La CDF viene calcolata con un'operazione di prefix sum sequenziale, mentre la LUT è generata secondo la nota formula normalizzata:

$$\text{LUT}(i) = \left\lfloor \frac{\text{cdf}[i] - \text{cdf}_{\min}}{N - \text{cdf}_{\min}} \cdot 255 \right\rfloor$$

Questa fase introduce un collo di bottiglia dovuto al trasferimento tra host e device, che interrompe il flusso puramente GPU-oriented dell'elaborazione.

3.5 Applicazione della LUT

Il kernel `applyLUT_global` applica la trasformazione LUT al canale Y. L'accesso è sequenziale ma potenzialmente non coalescente. Ogni thread aggiorna un solo valore del canale Y con accesso in lettura alla LUT:

```
Y[idx] = d_LUT[Y[idx]];
```

```cpp
__global__ void applyLUT_global(unsigned char* Y, const unsigned char* d_LUT, const int size)
{
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int stride = blockDim.x * gridDim.x;
    for (; idx < size; idx += stride) {
        unsigned char val = Y[idx];
        Y[idx] = d_LUT[val];
    }
}
```

Figure 7: applyLut_global

3.6 Conversione YUV → RGB

Similmente alla fase iniziale, il kernel `convertYUVtoRGB` ricombina i canali Y, U, V per ottenere nuovamente i valori RGB. Anche qui i thread scrivono su un array lineare `d_outputRGB`, accedendo a tre elementi consecutivi, ma in assenza di strutture `uchar3` si perde l'opportunità di sfruttare accessi coalescenti in scrittura.

3.7 Valutazione

Questa implementazione evidenzia i vantaggi del parallelismo rispetto alla CPU, ma presenta ancora limiti evidenti:

- Gli accessi alla memoria non sono coalescenti (né in lettura né in scrittura).
- Il calcolo della CDF e della LUT avviene su host, introducendo overhead.
- Le operazioni atomiche su memoria condivisa possono generare conflitti.

Si tratta quindi di una versione funzionale ma ancora migliorabile, che costituisce la base di partenza per successive ottimizzazioni strutturate.

4 Analisi della Versione `CUDA Ottimizzata`

La funzione `run_gpu_version` rappresenta una versione evoluta dell'implementazione CUDA base. Essa introduce una serie di ottimizzazioni mirate a migliorare le performance computazionali, ridurre la contesa tra thread e sfruttare al meglio la memoria globale della GPU. Le ottimizzazioni sono state applicate in vari punti critici del pipeline, dal layout dei dati fino al calcolo dell'istogramma.

4.1 Accesso coalescente alla memoria globale

Una delle modifiche principali riguarda il layout dei dati di input. Invece di utilizzare un array lineare di `unsigned char`, l'immagine è rappresentata come un array di strutture `uchar3`. Questo consente a ciascun thread di accedere ai tre

canali RGB di un pixel in modo contiguo, permettendo un accesso coalescente alla memoria globale:

```
uchar3 *d_inputRGB, *d_outputRGB;
cudaMalloc(&d_inputRGB, size * sizeof(uchar3));
cudaMemcpy(d_inputRGB, input, count: size * sizeof(uchar3),
    kind: cudaMemcpyHostToDevice);
```

Il kernel `convertRGBtoYUV_coalesced` utilizza direttamente `uchar3` per caricare il pixel in un'unica istruzione e calcolare le componenti YUV in modo più efficiente:

```
uchar3 pixel = input[idx];
unsigned char y_val, u_val, v_val;
rgbToYUV_device( r: pixel.x, g: pixel.y, b: pixel.z, [&] y_val, [&] u_val, [&] v_val);
```

4.2 Istogramma per warp su shared memory

Il kernel `computeHistogramWarped` migliora sensibilmente la gestione della memoria condivisa. Anziché un istogramma unico condiviso da tutto il blocco (come nella versione base), ogni warp dispone di un istogramma separato in shared memory:

```
__shared__ unsigned int warpHisto[MAX_WARPS][256];
```

Ogni thread aggiorna solo l'istogramma associato al proprio warp, riducendo la contesa sulle `atomicAdd` e migliorando la scalabilità del kernel. Alla fine, tutti gli istogrammi warp-local vengono combinati in un istogramma unico:

```
for (int w = 0; w < numWarps; ++w) {
    atomicAdd(&localHisto[bin], warpHisto[w][bin]);
}
```

4.3 Calcolo della CDF e LUT su host

Come nella versione base, il calcolo della CDF e della LUT viene effettuato su host. L'istogramma viene prima trasferito dalla memoria device alla memoria host, quindi processato con una semplice *prefix sum*. Nonostante ciò introduca una latenza dovuta alla copia, il carico computazionale è trascurabile rispetto al beneficio ottenuto dalla parallelizzazione delle fasi GPU-intensive:

```
cudaMemcpy(h_histogram,d_histogram, count: 256 * sizeof(int),
    kind: cudaMemcpyDeviceToHost);
```

L'uso della CPU per la costruzione della LUT consente di evitare l'uso della libreria Thrust, semplificando la dipendenza del progetto.

4.4 Applicazione della LUT e accessi efficienti

Il kernel `applyLUT_global` è identico alla versione base ma risulta più performante grazie a un accesso più regolare alla memoria e al fatto che l'input Y è stato generato con accesso coalescente. Questo migliora il comportamento della cache L2 della GPU.

4.5 Scrittura coalescente in output

La conversione finale da YUV a RGB viene eseguita tramite il kernel `convertYUVtoRGB_coalesced`, che utilizza nuovamente un array di `uchar3` per salvare l'output RGB. Anche in questo caso si garantisce coalescenza nella scrittura su memoria globale:

```
outputRGB[idx] = make_uchar3(r, g, b);
```

```
__global__ void convertYUVtoRGB_coalesced(const unsigned char* Y,
                                          const unsigned char* U,
                                          const unsigned char* V,
                                          uchar3* outputRGB,
                                          const int width,
                                          const int height)
{
    const unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    if (const unsigned int y = blockIdx.y * blockDim.y + threadIdx.y; x < width && y < height) {
        const unsigned int idx = y * width + x;
        const unsigned char y_val = Y[idx];
        const unsigned char u_val = U[idx];
        const unsigned char v_val = V[idx];

        unsigned char r, g, b;
        yuvToRGB_device(y_val, u_val, v_val, [&]r, [&]g, [&]b);
        outputRGB[idx] = make_uchar3(x:r, y:g, z:b); // Scrittura coalesced
    }
}
```

Figure 8: convertYUVToRGBCoalesced

4.6 Considerazioni tecniche e prestazionali

Le ottimizzazioni introdotte permettono di ridurre significativamente i colli di bottiglia identificati nella versione base:

- **Accesso coalescente**: miglior uso della banda di memoria globale grazie a `uchar3`.

- **Contesa ridotta**: uso di istogrammi warp-local in shared memory.
- **Efficienza generale**: ogni kernel è progettato per minimizzare il numero di accessi atomici e la sincronizzazione globale.

Pur mantenendo il calcolo CDF su CPU, questa versione è ben bilanciata in termini di complessità implementativa e performance. Costituisce un'ottima base per ulteriori miglioramenti, come l'integrazione di una *prefix sum* in device o l'intero pipeline GPU-resident.

5 Analisi delle Prestazioni

Per valutare l'efficacia delle ottimizzazioni implementate, sono stati eseguiti dei test di benchmark su tre immagini di dimensioni diverse. I risultati riportati nella Tabella 9 confrontano i tempi di esecuzione della versione sequenziale (`CPU`), della versione `CUDA ottimizzata` (denominata GPU A) e della versione `CUDA base` (GPU B). Viene inoltre calcolato il *speedup*, definito come:

$$\text{Speedup} = \frac{T_{\text{CPU}}}{T_{\text{GPU}}}$$

Immagine	Tempo CPU	Tempo GPU A	Tempo GPU B	Speedup A	Speedup B
immagine1	3520.31	7.47325	7.81837	471.055	450.261
immagine2	617.601	2.79197	3.1975	221.206	193.151
immagine3	570.484	2.46627	2.8705	231.314	198.741

Figure 9: Riepilogo delle performance per le tre immagini testate

5.1 Considerazioni

I dati dimostrano chiaramente un netto incremento di performance passando dalla CPU alla GPU, con uno *speedup* che varia da circa 193× fino a oltre 470×, a seconda dell'immagine e della versione CUDA utilizzata. In particolare:

- **GPU A (ottimizzata)** è sistematicamente più veloce rispetto a GPU B, grazie a:
  - accesso coalescente ai dati in memoria globale tramite `uchar3`;
  - utilizzo della *shared memory* per ridurre la contesa negli aggiornamenti dell'istogramma;
  - riduzione del numero di `atomicAdd` su variabili condivise.
- Le prestazioni della GPU risultano più stabili rispetto alla CPU, che mostra un'ampia variabilità nei tempi a seconda della dimensione e della complessità dell'immagine.

- Lo *speedup* più elevato è osservato su `immagine1`, probabilmente a causa della sua maggiore risoluzione, che evidenzia ancora di più i benefici del parallelismo massivo su GPU.

5.2 Osservazioni finali

L'analisi quantitativa dei tempi di esecuzione conferma il vantaggio significativo offerto dall'utilizzo della GPU rispetto all'implementazione CPU. L'ottimizzazione GPU A si dimostra superiore in tutti i casi testati, confermando l'importanza di una gestione efficiente della memoria e della contesa tra thread nei contesti di calcolo parallelo.

6 Conclusioni

Il progetto ha dimostrato l'efficacia dell'equalizzazione dell'istogramma come tecnica di miglioramento del contrasto, e ha permesso di confrontare concretamente un'implementazione sequenziale con due versioni parallele basate su CUDA. La transizione dalla CPU alla GPU ha comportato un notevole incremento delle prestazioni, grazie al parallelismo massivo e a un'attenta gestione della memoria.

Le ottimizzazioni introdotte nella versione GPU A, come l'accesso coalescente e l'uso di istogrammi per warp in `shared memory`, si sono rivelate determinanti nel migliorare l'efficienza rispetto alla versione base. I risultati confermano che, in scenari di elaborazione intensiva come il processamento di immagini ad alta risoluzione, l'uso della GPU è fortemente vantaggioso rispetto a un'esecuzione seriale.

Nel complesso, il lavoro svolto ha permesso di approfondire sia gli aspetti algoritmici dell'equalizzazione che le problematiche architetturali legate alla programmazione parallela.