

Parallel Computing Secondo Elaborato: Analisi delle Prestazioni di Histogram Equalization per la calibrazione del contrasto di immagini con CUDA

Giovanni Stefanini

giovanni.stefanini@edu.unifi.it

Abstract

L'elaborato presenta l'implementazione e l'analisi delle prestazioni di un algoritmo di Histogram Equalization, utilizzato per migliorare il contrasto delle immagini, sfruttando la parallelizzazione tramite CUDA. L'algoritmo è stato sviluppato in due versioni: una sequenziale eseguita su CPU e una parallela eseguita su GPU. I test sono stati condotti su immagini di diverse dimensioni, misurando i tempi di esecuzione e calcolando lo speedup della versione parallela rispetto a quella sequenziale.

1. Introduzione

L'Histogram Equalization è una tecnica di elaborazione delle immagini finalizzata al miglioramento del contrasto. Il suo principio si basa sulla redistribuzione dei livelli di intensità (sia in scala di grigi che a colori) in modo da ottenere una distribuzione più uniforme, incrementando il numero di pixel nelle aree meno rappresentate. L'obiettivo è trasformare l'istogramma originale in una distribuzione più equilibrata, utilizzando la funzione di distribuzione cumulativa (CDF - Cumulative Distribution Function) per rimappare i valori di intensità dei pixel. Questa tecnica trova applicazione in diversi ambiti, specialmente quando il contrasto dell'immagine è ridotto, con il fine di migliorarne la qualità visiva. Il risultato è un aumento del contrasto globale dell'immagine, come illustrato in Figura. 1

Il processo di Histogram Equalization segue i seguenti passi:

1. Calcolo dell'istogramma dell'immagine: conta il numero di pixel per ogni livello di intensità (ad esempio, in un'immagine in scala di grigi con 256 livelli, da 0 a 255).
2. Calcolo della funzione di distribuzione cumulativa (CDF): si sommano progressivamente le frequenze dell'istogramma per ottenere una funzione di trasformazione.
3. Normalizzazione della CDF: la CDF viene scalata per coprire l'intervallo di valori desiderato (ad esempio, 0-

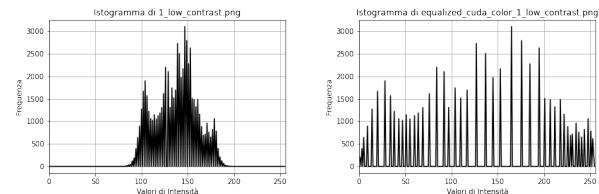


Figure 1: Fig. 1. Risultato dell'equalizzazione.

255 per immagini a 8 bit).

4. Rimappatura dei pixel: ogni valore di intensità originale viene sostituito con il corrispondente valore trasformato dalla CDF, ottenendo l'immagine equalizzata.

Obbiettivo principale dell'elaborato è analizzare e confrontare le performance di Histogram Equalization implementati con codice sequenziale e parallelo che sfrutta CUDA.

2. Implementazione

Si analizza ora l'implementazione del processo. Si inizia con la versione sequenziale, per poi esaminare la versione parallela in CUDA. Nel file CMakeLists.txt del progetto, è stato necessario abilitare la compilazione del codice C++, per la versione sequenziale, e del codice CUDA, per la versione parallela, utilizzando rispettivamente i comandi enable_language (CXX) e enable_language (CUDA). È da notare che per la gestione delle immagini è stato utilizzato OpenCV (Open

Source Computer Vision) che è una libreria open source per l'elaborazione delle immagini.

2.1. Versione Sequenziale

La versione sequenziale del codice implementa Histogram Equalization utilizzando OpenCV in C++ ed è contenuta nella funzione `histogram_equalization_seq`. Questa versione segue i classici passaggi dell'algoritmo, operando direttamente su una matrice OpenCV (Mat) per la gestione dell'immagine.

Controllo e conversione immagine in Scala di Grigi

L'algoritmo di equalizzazione viene applicato solo sulle immagini in scala di grigi, per questo motivo nel codice viene fatto preventivamente un controllo per verificare se l'immagine di input ha più di un canale (tipico delle immagini a colori). Se così fosse, viene convertita in scala di grigi utilizzando la funzione `cvtColor()` di OpenCV. Se invece l'immagine è già in scala di grigi, viene semplicemente clonata per evitare di modificarla direttamente.

Calcolo Istogramma

Una volta ottenuta l'immagine in scala di grigi, il passo successivo è costruire il suo istogramma, che rappresenta la frequenza di ogni valore di intensità (da 0 a 255). Per fare ciò, il codice scansiona l'intera immagine e conta quanti pixel hanno un determinato valore di luminosità. L'istogramma viene memorizzato in un array di 256 elementi, dove ogni posizione corrisponde al numero di pixel che possiedono quell'intensità. L'istogramma ci permette di capire la distribuzione dei livelli di luminosità nell'immagine e di definire una trasformazione che renda la distribuzione più uniforme.

Calcolo Funzione di Distribuzione Cumulativa (CDF)

Una volta ottenuto l'istogramma, si calcola la Funzione di Distribuzione Cumulativa (CDF). Questa funzione è costruita sommando progressivamente i valori dell'istogramma:

$$CDF[i] = CDF[i - 1] + hist[i]$$

Ciò significa che $CDF[i]$ conterrà il numero totale di pixel con intensità minore o uguale a i . La CDF permette di capire quanti pixel sono distribuiti sotto un certo livello di intensità e ci aiuta a costruire la nuova trasformazione dei valori di intensità.

Normalizzazione della CDF

Per garantire che i nuovi valori dei pixel siano distribuiti tra 0 e 255, la CDF deve essere normalizzata, in modo da ren-

denre il contrasto dell'immagine più uniforme. Si utilizza la formula:

$$\text{new_intensity}(i) = \frac{(CDF[i] - CDF_{\min}) \times 255}{\text{total_pixels} - CDF_{\min}}$$

Dove:

- $CDF[i]$ è il valore cumulativo per il livello di intensità i .
- CDF_{\min} è il primo valore non nullo della CDF (serve a evitare errori con immagini scure).
- total_pixels è il numero totale di pixel dell'immagine.

Questa operazione genera una lookup table, ovvero una tabella che assegna a ogni valore di intensità originale un nuovo valore equalizzato.

Applicazione della Trasformazione all'immagine

Infine, viene applicata la trasformazione: si scansiona nuovamente l'immagine e si sostituisce ogni pixel con il valore corrispondente trovato nella lookup table. In pratica, invece di ricalcolare la funzione per ogni pixel, si utilizza questa tabella precomputata, che permette di eseguire la sostituzione in modo molto più efficiente.

Osservazioni sulla versione sequenziale

Dall'esecuzione della versione sequenziale, si nota che il tempo di elaborazione diventa elevato per immagini di grandi dimensioni. Questo rallentamento è dovuto all'uso di cicli annidati su righe e colonne, che rendono il codice meno efficiente per immagini ad alta risoluzione. In particolare, 'Calcolo dell'Istogramma' e 'Applicazione della Trasformazione all'Immagine' richiedono un doppio ciclo (su righe e colonne), rendendoli ottimi candidati per la parallelizzazione mediante kernel CUDA, con l'obiettivo di migliorare le prestazioni. Anche il calcolo della CDF può essere ottimizzato con un approccio parallelo utilizzando CUDA, poiché si tratta di una somma progressiva dei valori dell'istogramma. In questo caso, l'uso di un Parallel Scan può accelerare il calcolo della CDF, riducendo ulteriormente i tempi di esecuzione.

Si passa quindi a una versione ottimizzata con CUDA, sfruttando la potenza della GPU per migliorare l'efficienza dell'elaborazione.

2.2. Versione Parallela

In questa implementazione, l'operazione è stata parallelizzata su GPU utilizzando CUDA per ottenere un significativo miglioramento delle prestazioni. In particolare, l'ottimizzazione si è concentrata sulla rimozione

dei doppi cicli `for` su righe e colonne dell’immagine e sull’efficientamento del calcolo della CDF. Per raggiungere questo obiettivo, i cicli `for` sono stati sostituiti da kernel CUDA, sfruttando l’architettura parallela della GPU. L’algoritmo si compone di tre fasi principali, ognuna delle quali viene eseguita su un kernel CUDA separato:

1. `computeHistogram`, kernel che calcola l’istogramma, ovvero conta quante volte ogni livello di grigio appare nell’immagine.
2. `computeCDF`, kernel che calcola la CDF, ovvero la somma cumulativa dei valori dell’istogramma.
3. `applyTransformation`, kernel che applica la trasformazione, ovvero mappa i vecchi valori di intensità ai nuovi valori normalizzati.

Il lancio di questi tre kernel e l’esecuzione del codice utile per effettuare l’equalizzazione dell’isogramma è contenuto tutto nella funzione `histogram_equalization_cuda`. Oltre a questi tre kernel principali, il codice include la gestione della memoria tra CPU e GPU, l’allocazione della memoria pinned per velocizzare le copie di dati, e il timing dell’esecuzione. Notare che la normalizzazione della CDF (calcolo della lookup table) avviene sulla CPU.

Kernel `computeHistogram`

Il primo kernel, `computeHistogram`, calcola l’istogramma dell’immagine in parallelo sfruttando la memoria condivisa. Ogni blocco di thread dispone di un’area di memoria shared in cui accumula un istogramma locale, riducendo così il traffico di memoria globale e migliorando l’efficienza. All’inizio, tutti i valori dell’istogramma locale vengono inizializzati a zero. Successivamente, i thread caricano i valori dei pixel dalla memoria globale nella shared memory e aggiornano l’istogramma locale utilizzando operazioni atomiche. Una volta completata questa fase, l’istogramma locale viene combinato con quello globale tramite un’operazione `atomicAdd`, garantendo che i risultati dei diversi blocchi vengano sommati correttamente.

Kernel `computeCDF`

Il secondo kernel, `computeCDF`, calcola la funzione di distribuzione cumulativa dell’istogramma mediante un’operazione di Scan Parallel. Questo passaggio è essenziale per ottenere una funzione di mappatura dei valori di intensità. I valori dell’istogramma vengono caricati in memoria condivisa per consentire un accesso più rapido e poi vengono elaborati utilizzando un algoritmo di riduzione, in cui ogni elemento viene aggiornato sommando il valore del predecessore con un certo offset. Dopo diverse iterazioni, ogni posizione dell’array conterrà la somma cumulativa dei valori precedenti, ottenendo così la CDF. È da notare che

l’offset raddoppia ($1, 2, 4, 8, \dots$) durante le iterazioni perché il calcolo della somma cumulativa segue una struttura gerarchica a livelli, dove ogni thread aggiorna il proprio valore sommando l’elemento precedente a una distanza crescente. Questa tecnica permette di eseguire la somma cumulativa in parallelo con un numero logaritmico di passaggi ($\log_2(256) = 8$), invece di fare 256 iterazioni in sequenza. Pertanto, il vantaggio della versione parallela risiede nel fatto che il calcolo della CDF richiede $O(\log N)$ operazioni, rispetto a $O(N)$ della versione sequenziale.

Kernel `applyTransformation`

Infine, il terzo kernel, `applyTransformation`, applica la trasformazione ai pixel dell’immagine utilizzando la lookup table precedentemente calcolata. I dati dell’immagine vengono caricati in memoria shared per migliorare l’accesso ai valori dei pixel. Ogni thread legge il valore di un pixel e lo sostituisce con il valore corrispondente nella lookup table, garantendo un accesso coalescente alla memoria globale per massimizzare l’efficienza della GPU.

Osservazioni sulla versione parallela

Per quanto riguarda la gestione della memoria, l’implementazione utilizza pinned memory per le strutture allocate sulla CPU, in modo da velocizzare i trasferimenti tra host e device. Inoltre, viene impiegato `cudaEvent` per misurare il tempo di esecuzione dei soli kernel, senza considerare il tempo impiegato dalle operazioni di copia della memoria. Questo permette di valutare con precisione le prestazioni dell’algoritmo parallelizzato. Dopo aver copiato il risultato dalla GPU alla CPU, tutte le allocazioni di memoria vengono rilasciate e gli eventi CUDA vengono distrutti per liberare risorse. L’implementazione sfrutta quindi in modo ottimale la memoria shared, le operazioni atomiche per l’aggiornamento dell’istogramma e la parallelizzazione della CDF, garantendo una significativa accelerazione rispetto a un’implementazione sequenziale su CPU.

3. Misurazione delle Prestazioni

Per valutare le prestazioni, i test sono stati eseguiti su otto immagini contenute nella cartella `img` del progetto, selezionate per rappresentare scenari in cui l’Histogram Equalization è particolarmente utile per migliorare il contrasto. Le immagini presentano risoluzioni differenti e sono numerate da 1 a 8, in ordine crescente di complessità nell’applicazione dell’equalizzazione. In particolare, i casi analizzati includono:

- immagine con basso contrasto (lc) 256x256,
- ambiente interno scuro (di) 1344x768,

- paesaggio nebbioso (f) 1280x813,
- immagini sovraesposte (o) 608x406 e 3680x2760,
- immagini sottoesposte (u) 1024x683 e 3680x2760,
- immagine ad alta risoluzione (hr) 3072x4096.

La selezione delle immagini è stata effettuata per analizzare sia lo speedup sia l'efficacia dell'equalizzazione in funzione della risoluzione e delle condizioni di illuminazione. Tutte le immagini provengono da fonti online, ad eccezione delle seconde immagini sottoesposte e sovraesposte, che sono state acquisite con una Polaroid SNAP. Le immagini vengono riportate in Figura 2.

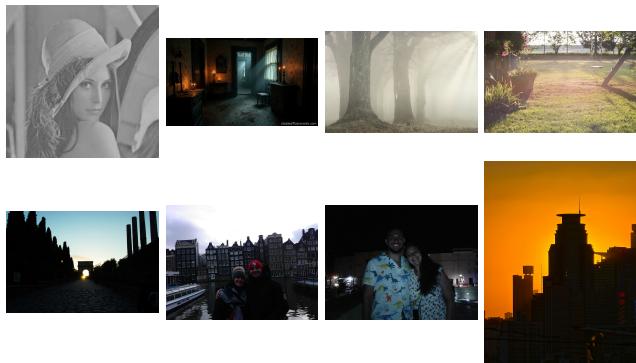


Figure 2: Immagini non equalizzate selezionate per i test.

3.1. Specifiche tecniche

Il computer utilizzato per eseguire le misurazioni ha le seguenti specifiche:

Specifiche Hardware

CPU: Intel (R) Core (TM) i7-7700HQ Kabylake
Frequenza: 2.8 GHz
RAM: 16 GB
GPU: NVIDIA GeForce GTX 1050

Specifiche Software

SO: Microsoft Windows 10 Home
Compilatore CUDA: nvcc 12.8.61
Compilatore Host: MSVC 1942
IDE: CLion 2024.2.2

3.2. Metriche di analisi

Di seguito sono riportate le metriche impiegate per valutare e confrontare le performance delle versioni sequenziale e parallela di Histogram Equalization.

Tempo di Esecuzione

Per misurare il tempo di esecuzione delle due versioni dell'algoritmo, è stato implementato un cronometro utilizzando `std::chrono`, una libreria standard di C++ che consente una misurazione precisa del tempo. Nel codice, il tempo viene registrato immediatamente prima (`start_`) e dopo (`end_`) l'esecuzione delle funzioni `histogram_equalization_seq` e `histogram_equalization_cuda`. In questo modo, calcolando la differenza tra `start_` ed `end_`, è possibile ottenere il tempo effettivo di esecuzione delle sole funzioni. È importante notare che, nel tempo misurato per la versione CUDA con `std::chrono`, sono inclusi anche gli overhead di comunicazione tra CPU e GPU, oltre al tempo necessario per il cleanup della memoria. Per questo motivo, per isolare il tempo di esecuzione dei soli kernel, escludendo le operazioni di copia di memoria, all'interno della funzione `histogram_equalization_cuda` è stato utilizzato `cudaEventRecord` per una misurazione più accurata.

Speedup

Lo Speedup (S) è una metrica che misura il miglioramento delle prestazioni di un algoritmo parallelo rispetto alla sua versione sequenziale. Esso è definito come il rapporto tra il tempo di esecuzione della versione sequenziale e quello della versione parallela:

$$\text{Speedup} = \frac{T_{\text{sequenziale}}}{T_{\text{parallelo}}}$$

dove $T_{\text{sequenziale}}$ è il tempo di esecuzione della versione sequenziale dell'algoritmo e $T_{\text{parallelo}}$ è il tempo di esecuzione della versione parallela. Quindi, lo Speedup indica quante volte l'algoritmo parallelo è più veloce di quello sequenziale. L'interpretazione che possiamo dare allo Speedup è la seguente:

- Se $S > 1$, allora la versione parallela è più veloce della sequenziale.
- Se $S \approx 1$, allora non c'è un miglioramento significativo.
- Se $S < 1$, la versione parallela è più lenta (per overhead o poca parallelizzabilità).

Quando si utilizza CUDA per la parallelizzazione, è importante distinguere tra Speedup Teorico e Speedup Effettivo. Lo Speedup Teorico (S_{kernel}) viene calcolato utilizzando il tempo di esecuzione dei soli kernel CUDA (T_{kernel}), escludendo i costi di trasferimento dati tra CPU e GPU. Questo tipo di Speedup viene utilizzato per analizzare quanto bene scala il calcolo sulla GPU e per ottimizzare i kernel. D'altra parte, lo Speedup Effettivo (S_{cuda}) include il tempo totale di esecuzione della versione parallela (T_{cuda}), comprensivo dei tempi di trasferimento dei dati tra la CPU e la GPU.

Questo Speedup fornisce una misura più realistica di quanto velocemente l'intero programma viene eseguito rispetto alla versione CPU.

In seguito, verranno calcolati entrambi gli Speedup. Se S_{cuda} risulta elevato ma S_{kernel} è basso, ciò indica che l'overhead dei trasferimenti non è efficiente e deve essere ottimizzato.

4. Risultati

Le immagini risultanti da Histogram Equalization sequenziale sono mostrate in Figura 3, mentre quelle ottenute con Histogram Equalization parallelo sono illustrate in Figura 4.

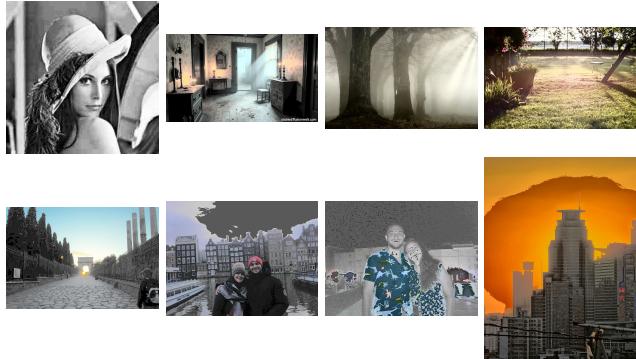


Figure 3: Immagini equalizzate con versione sequenziale.

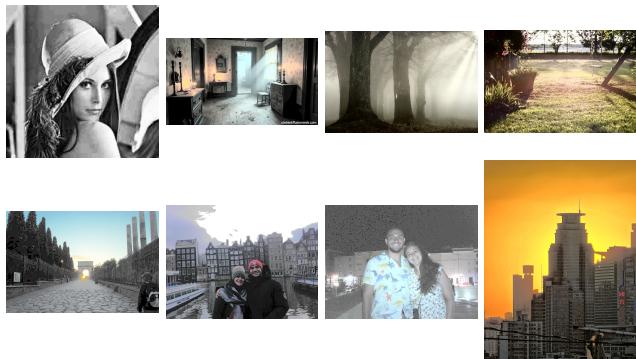


Figure 4: Immagini equalizzate con versione CUDA.

Confrontando le immagini, si nota che il risultato dell'equalizzazione è lo stesso per le immagini a risoluzione più bassa, mentre possiamo notare come le immagini a risoluzione più alta (le 3 immagini in basso a destra di Figura 3 e Figura 4) risultano essere meglio equalizzate dalla versione parallela che sfrutta CUDA.

4.1. Analisi dei tempi di Esecuzione

I tempi di esecuzione dell'algoritmo sequenziale e parallelo sono stati misurati sulle 8 immagini di test che hanno

risoluzioni differenti e rappresentano condizioni di illuminazione differenti. È da notare che per la versione parallela viene riportato sia il tempo di esecuzione dei soli kernel (T_{kernel}) sia del tempo totale di esecuzione della versione parallela (T_{cuda}), comprensivo dei tempi di trasferimento dei dati tra la CPU e la GPU. I risultati sono riportati nella Tabella 1 seguente:

Immagine	$T_{sequenziale}$ (ms)	T_{kernel} (ms)	T_{cuda} (ms)
lc 256x256	2.1704	0.233248	89.3771
di 1344x768	34.5987	0.580288	78.435
f 1280x813	36.0089	0.575712	91.6379
o 608x406	8.3431	0.450464	89.9615
u 1024x683	22.8673	0.63056	89.523
o 3680x2760	359.179	3.46342	599.038
u 3680x2760	335.684	3.52854	636.197
hr 3072x4096	421.064	4.29763	564.916

Table 1: Tempi di esecuzione per immagini con risoluzione e illuminazioni diverse.

Analizzando i tempi di esecuzione della versione parallela, ovvero quelli relativi ai kernel CUDA, si osserva un netto miglioramento rispetto alla versione sequenziale, indicando un calcolo significativamente ottimizzato. Tuttavia, considerando il tempo totale di esecuzione della versione parallela, si nota un incremento significativo, suggerendo che una parte rilevante del tempo viene impiegata per le operazioni di trasferimento dati tra Host e Device.

4.2. Analisi dello Speedup

Proseguendo nell'analisi precedente, sono stati calcolati gli speedup in relazione ai tempi di esecuzione precedentemente riportati. Nella Tabella 2, è presentato come varia lo Speedup Teorico (S_{kernel}) e lo Speedup Effettivo (S_{cuda}) in base alla risoluzione e l'illuminazione dell'immagine.

Immagine	S_{kernel}	S_{cuda}
lc 256x256	9.31	0.024
di 1344x768	59.6	0.44
f 1280x813	62.5	0.39
o 608x406	18.5	0.093
u 1024x683	36.3	0.26
o 3680x2760	103.7	0.6
u 3680x2760	95.13	0.53
hr 3072x4096	97.99	0.75

Table 2: Speedup per immagini con risoluzione e illuminazioni diverse.

L'analisi della Tabella 2 mostra che, osservando S_{kernel} , il kernel CUDA risulta significativamente più veloce rispetto all'esecuzione sequenziale, con uno speedup che

aumenta all'aumentare della dimensione dell'immagine. Questo evidenzia l'efficacia della parallelizzazione su GPU e la sua capacità di scalare con l'incremento dei dati. D'altro canto, analizzando S_{cuda} , si nota che l'intero programma in CUDA risulta più lento rispetto alla versione sequenziale. Il fattore limitante principale è il tempo totale di esecuzione, dominato dall'overhead della comunicazione tra CPU e GPU. Per immagini di piccole dimensioni, il costo del trasferimento dati annulla completamente i benefici del calcolo parallelo. In sintesi il kernel CUDA è altamente efficiente in quanto lo speedup del kernel varia da 9x a 100x rispetto alla CPU, dimostrando l'efficacia della computazione parallela. Però si ha un significativo overhead CPU-GPU in quanto per immagini di piccole dimensioni, il tempo totale di esecuzione in CUDA risulta superiore a quello della versione sequenziale. Si evince che l'uso della GPU è vantaggioso per immagini grandi, infatti per risoluzioni elevate, come 3680x2760 o 3072x4096, l'overhead incide meno e si inizia a ottenere un miglioramento complessivo.

4.3. Analisi della Occupancy

L'Occupancy serve a stimare l'efficienza dell'uso della GPU nel codice CUDA. In particolare, l'occupancy di un kernel CUDA rappresenta il rapporto tra il numero di thread attivi e il numero massimo teorico che la GPU può gestire simultaneamente. Un'occupancy elevata, tra il 60% e il 100%, indica una buona parallelizzazione e un utilizzo efficiente delle risorse della GPU. Se il valore si attesta tra il 30% e il 60%, potrebbe esserci margine di miglioramento, anche se l'impatto sulle prestazioni dipende dal tipo di workload. Quando l'occupancy è inferiore al 30%, significa che le risorse non vengono sfruttate al meglio e ciò può limitare l'efficacia dell'elaborazione parallela. Tuttavia, è importante notare che un'occupancy del 100% non è sempre necessaria per ottenere buone prestazioni. Alcuni kernel possono funzionare in modo ottimale anche con un'occupancy più bassa, soprattutto se ogni thread richiede un uso intensivo di risorse come registri e memoria condivisa.

Per stimare l'occupancy della GPU, sono stati inseriti degli output nel codice per raccogliere informazioni utili a questo scopo. Il calcolo è stato effettuato utilizzando il CUDA Occupancy Calculator, disponibile online alla pagina [cuda-calculator](#). Questo strumento richiede alcuni parametri chiave, tra cui la Compute Capability, il numero di Threads per blocco, i Registri per thread e la Shared memory per blocco. Per determinare il numero di thread per blocco, è stata verificata la dimensione del blocco impostata nel codice, che è pari a 16x16, ovvero 256 thread per blocco. I dati hardware utilizzati per l'analisi sono i seguenti:

CPU: Intel (R) Core (TM) i7-7700HQ Kabylake

Frequenza: 2.8 GHz
 RAM: 16 GB
 GPU: NVIDIA GeForce GTX 1050
 Compute Capability: 6.1
 SMs (Streaming Multiprocessors): 5
 Max Registers per block: 65536
 Shared memory per block: 49152 bytes
 Threads per SM: 2048
 Thread per blocco = 256

L'occupancy è stata calcolata per ciascun kernel, raccogliendo le seguenti informazioni:

Registri per thread K1: 10
 Shared memory per blocco K1: 1088 bytes
 Registri per thread K2: 12
 Shared memory per blocco K2: 1024 bytes
 Registri per thread K3: 8
 Shared memory per blocco K3: 64 bytes

I risultati mostrano che, per tutti e tre i kernel, si ottiene un'occupancy del 100%, il che significa che ogni Streaming Multiprocessor (SM) sta eseguendo il massimo numero possibile di warps attivi. Questo indica che la GPU è sfruttata al massimo delle sue capacità e che i kernel stanno funzionando in modo efficiente, senza la necessità di ulteriori ottimizzazioni.

5. Conclusioni

L'implementazione parallela di Histogram Equalization con CUDA offre un significativo miglioramento delle prestazioni rispetto alla versione sequenziale, sia in termini di tempi di calcolo che di qualità visiva dei risultati, soprattutto per immagini di grandi dimensioni. Tuttavia, considerando il tempo di esecuzione complessivo, che include anche l'overhead dovuto ai trasferimenti di dati tra CPU e GPU, il vantaggio della versione parallela potrebbe ridursi, rendendo necessario valutare attentamente l'impatto delle operazioni di comunicazione sulla velocità complessiva dell'algoritmo.

6. Riferimenti

- Slides corso "Parallel Computing", Prof. Marco Bertini
- NVIDIA. CUDA C Programming Guide. Disponibile su: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>
- Histogram equalization, en.wikipedia.org

Codice disponibile al seguente GitHub.