Integral Image

Leonardo Del Bene
leonardo.delbene@edu.unifi.it

Abstract

L'algoritmo "Integral Image" è un metodo efficiente per il calcolo di regioni rettangolari all'interno di un'immagine, ed è ampiamente utilizzato in computer vision per applicazioni come il rilevamento di caratteristiche e la segmentazione delle immagini. Il presente lavoro esplora una versione parallela dell'algoritmo, implementata utilizzando CUDA. L'implementazione parallela mira a ridurre i tempi di calcolo necessari per il processamento di grandi volumi di dati, sfruttando l'architettura massivamente parallela delle GPU moderne. Attraverso un'analisi comparativa delle performance tra l'implementazione sequenziale e quella parallela, si dimostra come l'uso di CUDA permetta un miglioramento significativo in termini di velocità di esecuzione, in scenari di grandi dimensioni.

1. Introduzione

L'**Integral Image**, o Somma Integrale, è una trasformazione fondamentale nell'elaborazione delle immagini, utilizzata per accelerare il calcolo delle somme su sotto-regioni di un'immagine. Introdotta inizialmente per applicazioni come il rilevamento degli oggetti (ad esempio, nell'algoritmo di Viola-Jones per il face detection), questa tecnica consente di calcolare rapidamente la somma dei pixel all'interno di qualsiasi finestra rettangolare in tempo costante, indipendentemente dalle dimensioni dell'area selezionata.

L'idea alla base dell'Integral Image è quella di costruire una **matrice cumulativa** in cui ogni elemento rappresenta la somma di tutti i pixel in alto a sinistra rispetto alla posizione corrente. Una volta costruita questa struttura, il calcolo della somma su una regione arbitraria può essere ottenuto con sole quattro operazioni, indipendentemente dalle dimensioni della finestra, migliorando significativamente le prestazioni rispetto a un approccio naïve.

Nel contesto di calcoli su immagini di grandi dimensioni, l'approccio sequenziale tradizionale risulta limitante a causa della dipendenza tra i dati e della crescita della complessità computazionale. Per superare queste limitazioni, l'uso della **parallelizzazione su GPU** con CUDA rappresenta una soluzione efficace, sfruttando il parallelismo massivo offerto dalle moderne schede grafiche per ridurre drasticamente i tempi di esecuzione.

In questo progetto, verrà analizzato il calcolo dell'Integral Image sia nella sua versione sequenziale su CPU che nella sua versione parallela su GPU con CUDA. Il confronto sarà basato su metriche di performance come **tempo di esecuzione, speedup ed efficienza**, evidenziando i vantaggi e le eventuali sfide della parallelizzazione.

Vantaggi L'uso dell'Integral Image presenta diversi vantaggi:

- **Computazione della somma in tempo** $O(1)$: dopo la pre-elaborazione, qualsiasi somma regionale viene calcolata istantaneamente.
- **Efficienza nel rilevamento di pattern**: ampiamente utilizzata in computer vision per il rilevamento di caratteristiche.
- **Facilità di implementazione**: richiede solo operazioni aritmetiche elementari e una struttura di dati della stessa dimensione dell'immagine originale.

L'implementazione sequenziale dell'Integral Image è efficiente, ma per immagini di grandi dimensioni può essere migliorata sfruttando il parallelismo. Nelle sezioni successive analizzeremo l'implementazione dell'algoritmo su **GPU con CUDA**, confrontando le prestazioni con la versione sequenziale.

2. Algoritmo di Integral Image

2.1. Versione sequenziale

L'**Integral Image** (o *Somma Integrale*) è una trasformazione utile nell'elaborazione delle immagini per calcolare efficientemente la somma dei pixel su sotto-regioni rettangolari.

L'idea principale è costruire una matrice integrale in cui ogni elemento (i, j) rappresenta la somma cumulativa di tutti i pixel dell'immagine originale dalla posizione $(0, 0)$ fino a (i, j). La formula utilizzata per il calcolo è:

$$\text{integral}(i,j) = \text{image}(i,j) + \text{integral}(i-1,j) + \text{integral}(i,j-1) - \text{integral}(i-1,j-1) \quad (1)$$

dove:

- $\text{image}(i,j)$ è il valore del pixel nella posizione (i,j).
- $\text{integral}(i-1,j)$ rappresenta la somma cumulativa della riga precedente.
- $\text{integral}(i,j-1)$ rappresenta la somma cumulativa della colonna precedente.
- $\text{integral}(i-1,j-1)$ viene sottratto per evitare un doppio conteggio.

2.2. Versione parallela

L'implementazione dell'algoritmo di *Integral Image* su GPU si basa sullo **Scan** delle righe della matrice di input. Lo *scan* di un vettore produce in output un vettore della stessa dimensione, dove il *j-esimo* elemento è formato dalla somma di tutti gli elementi precedenti. La mia implementazione parallela dell'Integral Image prevede i seguenti passaggi:

1. Eseguire un primo *scan* su ogni riga della matrice di input.
2. Applicare una trasposizione della matrice risultante.
3. Eseguire un secondo *scan* su ogni riga della matrice trasposta.
4. Applicare una seconda trasposizione della matrice.

Tutte le operazioni citate nell'elenco vengono eseguite su GPU.

2.2.1 Algoritmo di Scan

L'operazione di *scan*, detta anche *prefix sum*, trasforma un vettore di input in un vettore di output della stessa dimensione, in cui ogni elemento nella posizione j è dato dalla somma di tutti gli elementi precedenti nel vettore di input. Formalmente, dato un vettore di input:

$$x = [x_0, x_1, x_2, \ldots, x_{n-1}]$$

l'operazione di scan genera il vettore:

$$y = [x_0, x_0 + x_1, \ldots, x_0 + x_1 + \cdots + x_{n-1}]$$

Questo algoritmo è particolarmente utile in vari ambiti della programmazione parallela, come il calcolo di istogrammi, algoritmi di ordinamento e altre operazioni su array. L'implementazione parallela sfrutta una strategia in due fasi:

- **Up-sweep (riduzione)**: si costruisce la somma parziale degli elementi usando un pattern binario.
- **Down-sweep (propagazione)**: si distribuiscono i risultati parziali per ottenere la somma prefissa corretta.

Di seguito è riportato il codice CUDA per l'implementazione parallela dello scan su GPU:

```
__global__ void scan_kernel(int *d_in, int *d_out,
                            int *block_sums, int m) {
    extern __shared__ int temp[];
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    int tid = threadIdx.x;

    // Caricamento in memoria condivisa
    if (idx < m) {
        temp[tid] = d_in[idx];
    } else {
        temp[tid] = 0;
    }
    __syncthreads();

    // Up-sweep (costruzione della somma parziale)
    for (int offset = 1; offset < blockDim.x;
                                  offset *= 2) {
        int index = (tid + 1) * offset * 2 - 1;
        __syncthreads();
        if (index < blockDim.x) {
            temp[index] += temp[index - offset];
        }
    }

    // Down-sweep (propagazione della somma)
    for (int offset = blockDim.x / 2; offset > 0;
                                  offset /= 2) {
        int index = (tid + 1) * offset * 2 - 1;
        __syncthreads();
        if (index + offset < blockDim.x) {
            temp[index + offset] += temp[index];
        }
    }
    __syncthreads();

    // Salvataggio del valore finale per la correzione
       tra blocchi
    if (tid == blockDim.x - 1) {
        block_sums[blockIdx.x] = temp[tid];
    }

    // Scrittura dei risultati nell'output globale
    if (idx < m) {
        d_out[idx] = temp[tid];
    }
}
```

Si usa `extern __shared__ int temp[]` per gestire i dati localmente all'interno di ogni blocco. Ogni thread car-

ica un elemento di `d_in` in `temp`, gestendo il caso in cui `idx >= m` con un valore di default 0.

Nella fase di **Up-sweep** si usa un pattern binario per costruire le somme parziali all'interno di ciascun blocco. Ad ogni iterazione, gli elementi a intervalli crescenti vengono sommati.

Nella fase di **Down-sweep** si distribuisce la somma accumulata in modo da ottenere la somma prefissa per ogni elemento. Ogni thread aggiorna i valori della memoria condivisa propagando i risultati in modo parallelo.

L'ultimo valore calcolato in ogni blocco viene salvato in `block_sums` per essere poi usato per combinare i risultati dei diversi blocchi in una fase successiva. Questa implementazione è ottimizzata per GPU grazie all'uso della memoria condivisa ed utilizza un accesso coalessed alla memoria globale, permettendo di ottenere un'efficace parallelizzazione dell'operazione di scan.

Dettagli implementativi scan parallelo: Per eseguire uno scan di una riga della matrice utilizzando più blocchi CUDA, è necessario garantire che i risultati ottenuti da ciascun blocco siano coerenti. In particolare, al termine dello scan parallelo di ciascun blocco, viene salvato l'ultimo elemento del vettore parziale di ogni blocco, che rappresenta la somma cumulativa finale calcolata per quel blocco. Questi valori vengono memorizzati in un vettore denominato `block_sums`, la cui lunghezza è pari al numero di blocchi utilizzati per il calcolo della riga.

Successivamente, viene eseguito uno scan parallelo sul vettore `block_sums` per ottenere la somma cumulativa globale. Questo passaggio è essenziale per garantire che gli effetti della somma cumulativa siano propagati correttamente tra i diversi blocchi. Dopo che lo scan sul vettore `block_sums` è stato completato, ogni elemento della riga della matrice viene sommato con il rispettivo valore presente nel vettore `block_sums`. In questo modo, la somma cumulativa finale per ogni elemento della riga viene correttamente calcolata.

Queste operazioni sono particolarmente importanti quando la matrice di input ha più di 1024 colonne, poiché 1024 è il numero massimo di thread che possono essere eseguiti all'interno di un singolo blocco CUDA. Utilizzando più blocchi per gestire una singola riga della matrice, è possibile superare questa limitazione e gestire matrici di dimensioni più grandi, garantendo comunque una corretta esecuzione parallela dello scan.

Input e Output dell'algoritmo di Integral Image: Consideriamo una matrice 8×8 di valori interi compresi tra 0 e 9.

Matrice iniziale:

$$\begin{pmatrix} 1 & 4 & 6 & 7 & 7 & 5 & 7 & 9 \\ 6 & 5 & 0 & 2 & 4 & 7 & 4 & 5 \\ 8 & 9 & 7 & 3 & 5 & 5 & 0 & 6 \\ 6 & 7 & 0 & 3 & 7 & 7 & 8 & 7 \\ 5 & 8 & 2 & 2 & 8 & 0 & 5 & 0 \\ 2 & 7 & 1 & 3 & 0 & 5 & 9 & 3 \\ 1 & 8 & 1 & 0 & 6 & 4 & 3 & 3 \\ 5 & 9 & 8 & 5 & 1 & 9 & 2 & 8 \end{pmatrix}$$

Matrice finale:

$$\begin{pmatrix} 1 & 5 & 11 & 18 & 25 & 30 & 37 & 46 \\ 7 & 16 & 22 & 31 & 42 & 54 & 65 & 79 \\ 15 & 33 & 46 & 58 & 74 & 91 & 102 & 122 \\ 21 & 46 & 59 & 74 & 97 & 121 & 140 & 167 \\ 26 & 59 & 74 & 91 & 122 & 146 & 170 & 197 \\ 28 & 68 & 84 & 104 & 135 & 164 & 197 & 227 \\ 29 & 77 & 94 & 114 & 151 & 184 & 220 & 253 \\ 34 & 91 & 116 & 141 & 179 & 221 & 259 & 300 \end{pmatrix}$$

3. Setup Esperimenti

In tutti gli esperimenti condotti, è stato utilizzato come input una matrice quadrata, generata con valori casuali nell'intervallo tra 0 e 255, al fine di simulare un'immagine in scala di grigi con un singolo canale.

Per quanto riguarda l'hardware utilizzato, il codice sequenziale e il codice CUDA sono stati eseguiti sul server del Dipartimento dell'Informazione, dotato di due schede grafiche NVIDIA RTX A2000.

Per valutare il guadagno in termini di tempo di esecuzione tra la versione sequenziale e quella parallela basata su CUDA, sono stati eseguiti numerosi test utilizzando matrici di dimensioni crescenti: $10000x10000$, $15000x15000$, $25000x25000$, $40000x40000$ e $60000x60000$. Ogni test è stato eseguito 5 volte, e per trarre delle conclusioni è stato considerato il tempo medio di esecuzione, al fine di ottenere una misura più accurata delle prestazioni.

4. Risultati Ottenuti

In questo paragrafo sono riportati i risultati ottenuti in termini di tempo di esecuzione per la versione sequenziale dell'algoritmo, su diverse dimensioni della matrice. I tempi di esecuzione sono stati misurati per ogni test e sono stati calcolati i tempi medi su 10 esecuzioni per ogni configurazione.

4.1. Risultati versione sequenziale

Nella tabella 1 sono riportati i tempi di esecuzione riscontrati dall'algoritmo sequenziale. La tabella mostra i

Dimensione Matrice	Run 1 (ms)	Run 2 (ms)	Run 3 (ms)	Run 4 (ms)	Run 5 (ms)	Tempo Medio (ms)
10000×10000	2067	2062	2064	2061	2062	2063
15000×15000	4627	4668	4630	4634	4630	4632.8
25000×25000	12928	12902	12889	12867	12851	12887.4
40000×40000	32920	32892	32917	32988	32892	32922.2
60000×60000	73931	74236	74362	74905	74624	74412.6

Table 1. Tempi di esecuzione (ms) per le diverse dimensioni della matrice eseguite 5 volte ciascuna.

Dimensione Matrice	Run 1 (ms)	Run 2 (ms)	Run 3 (ms)	Run 4 (ms)	Run 5 (ms)	Tempo Medio (ms)
10000×10000	938	909	830	578	802	811
15000×15000	742	919	851	802	926	848
25000×25000	950	919	1275	954	1086	1037
40000×40000	1645	1422	1804	1591	1491	1591
60000×60000	848	752	1175	765	963	901

Table 2. Tempi di esecuzione (ms) per le diverse dimensioni della matrice eseguite 5 volte ciascuna nella versione CUDA.

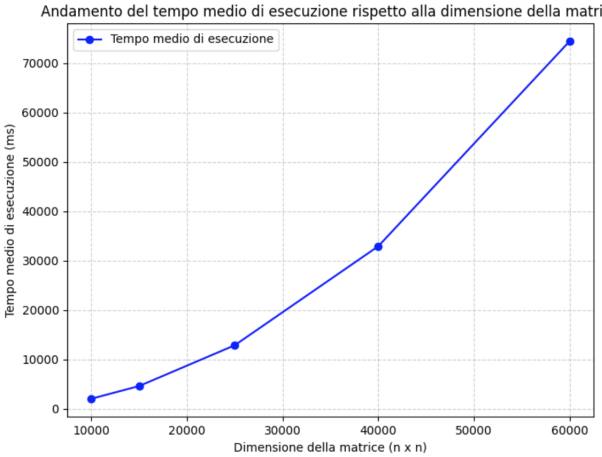

Figure 1. Tempi di esecuzione versione sequenziale rispetto alla dimensione della matrice di input

tempi delle singole esecuzioni insieme al tempo medio di esecuzione per ogni configurazione. In totale, per ciascuna configurazione, sono stati eseguiti 5 test.

Dai risultati nella tabella 1 e dalla figura 1 si osserva che all'aumentare della dimensione della matrice cresce anche il tempo di esecuzione, come previsto. In particolare, il tempo medio di esecuzione aumenta in maniera significativa con il crescere della dimensione della matrice, evidenziando la complessità computazionale dell'algoritmo sequenziale.

4.2. Risultati versione CUDA

Nella tabella 2 sono riportati i tempi di esecuzione dell'algoritmo nella versione CUDA. Per ogni configurazione sono stati eseguiti 5 test.

Dai risultati nella tabella 2 e dalla figura 2 si nota che l'implementazione CUDA permette di ridurre significativamente i tempi di esecuzione rispetto alla versione sequenziale. Nonostante alcune variazioni tra i diversi test, il tempo medio di esecuzione mostra un netto miglioramento rispetto ai valori ottenuti nella tabella 1. Questo conferma l'efficacia del calcolo parallelo nella gestione di matrici di grandi dimensioni, evidenziando il vantaggio dell'accelerazione GPU rispetto all'elaborazione sequenziale su CPU.

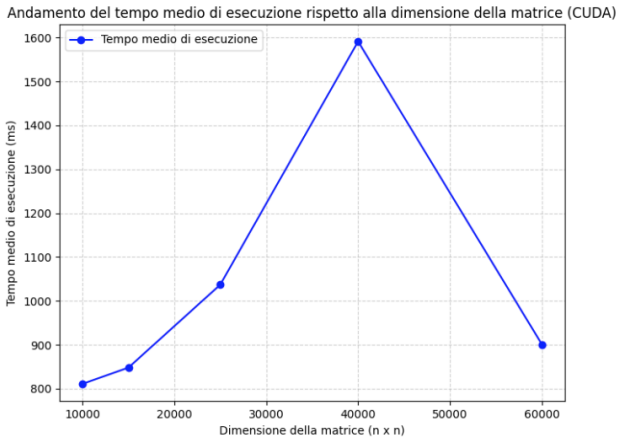

Figure 2. Tempi di esecuzione versione CUDA rispetto alla dimensione della matrice di input

4.3. SpeedUp

In questo paragrafo vogliamo concentrarsi sull'analisi dello *speedup*, calcolato come il rapporto tra il tempo di esecuzione nella versione sequenziale e il tempo di esecuzione nella versione CUDA. I dati ottenuti sono riportati nella tabella 3 Dai risultati emerge chiaramente un significativo miglioramento delle prestazioni con l'uso della GPU. Analizzando i valori di speedup (in tabella 3 e nella figura 3), si osserva che per la configurazione più piccola (10000×10000), il miglioramento è relativamente modesto,

Dim Matrice	V. Seq (ms)	V. CUDA(ms)	Speedup
10000×10000	2063	811	2.54
15000×15000	4632.8	848	5.46
25000×25000	12887.4	1037	12.43
40000×40000	32922.2	1591	20.71
60000×60000	74412.6	901	82.58

Table 3. Calcolo dello speedup come rapporto tra i tempi medi della versione sequenziale e della versione CUDA.

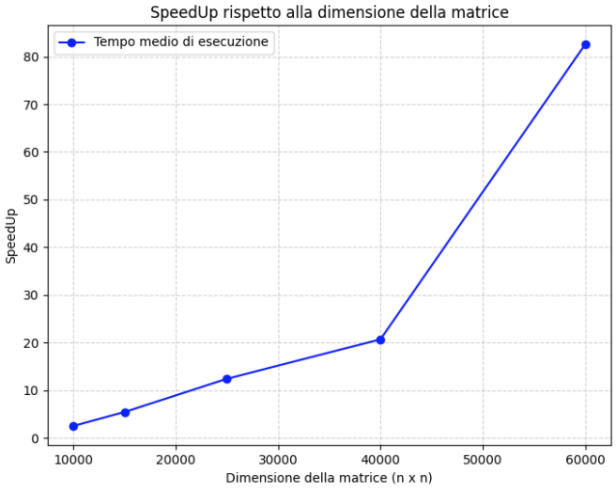

Figure 3. Grafico dello speedup in funzione della dimensione della matrice di input

con un fattore di circa 2.54. Questo è prevedibile, poiché per matrici di dimensioni ridotte l'overhead della gestione dei thread CUDA può limitare i benefici del parallelismo.

All'aumentare della dimensione della matrice, lo speedup cresce progressivamente, raggiungendo valori sempre più elevati. Per la matrice di dimensioni 25000×25000 si ottiene già un'accelerazione superiore a $12\times$, mentre con la configurazione 40000×40000 lo speedup arriva a circa $20.71\times$. Questo comportamento è dovuto al fatto che, con matrici più grandi, l'architettura parallela della GPU è sfruttata in modo più efficiente, ammortizzando l'overhead iniziale e massimizzando il parallelismo.

Il caso più estremo, relativo alla matrice 60000×60000, mostra uno speedup impressionante di $82.58\times$. Questo risultato evidenzia il grande vantaggio delle GPU nell'elaborazione di grandi volumi di dati, dove la capacità di eseguire calcoli in parallelo su migliaia di core consente di ottenere guadagni di prestazioni molto elevati rispetto alla CPU.

In sintesi, i risultati confermano che l'uso di CUDA permette un'accelerazione significativa, soprattutto per matrici di grandi dimensioni, dove il parallelismo massiccio della GPU può essere sfruttato al massimo. Per matrici più piccole, l'overhead della gestione dei thread e della comunicazione con la GPU può ridurre i benefici del parallelismo, ma man mano che la dimensione aumenta, il guadagno in termini di tempo di esecuzione diventa sempre più evidente.

5. Conclusioni

In questo lavoro abbiamo analizzato le prestazioni di un algoritmo di **Integral Image** implementato in versione sequenziale e parallela tramite CUDA. I risultati ottenuti mostrano chiaramente il vantaggio dell'approccio parallelo, specialmente per matrici di grandi dimensioni.

Dai dati ottenuti nella versione sequenziale, riportati nella tabella 1, emerge che il tempo di esecuzione cresce significativamente all'aumentare delle dimensioni della matrice. La complessità computazionale dell'algoritmo sequenziale rende evidente la difficoltà di gestire matrici molto grandi utilizzando una sola CPU.

Invece, l'implementazione parallela tramite CUDA, presentata nella tabella 2, permette di ridurre drasticamente i tempi di esecuzione rispetto alla versione sequenziale. La GPU, grazie alla sua architettura massicciamente parallela, consente di gestire in modo efficiente matrici di grandi dimensioni, con un aumento delle prestazioni che cresce proporzionalmente alla dimensione della matrice. Questo è evidente dalla tabella 3 e dalla figura 3, dove lo *speedup* aumenta notevolmente passando da matrici più piccole a quelle più grandi.

Per matrici di dimensioni più contenute, l'overhead del parallelismo, dovuto alla gestione dei thread e alla comunicazione con la GPU, limita i guadagni di prestazioni. Tuttavia, con l'aumento della dimensione della matrice, i vantaggi derivanti dal parallelismo si manifestano chiaramente, portando a miglioramenti significativi nelle prestazioni, come nel caso della matrice 60000×60000, dove si osserva uno speedup di circa $82.58\times$.

In conclusione, l'adozione di CUDA rappresenta una scelta vincente per l'elaborazione di matrici di grandi dimensioni, in quanto consente di sfruttare appieno la potenza di calcolo della GPU. Per applicazioni che richiedono l'elaborazione di grandi volumi di dati, l'approccio parallelo può portare a riduzioni sostanziali dei tempi di esecuzione, rendendo possibile affrontare compiti computazionali complessi in tempi significativamente più brevi rispetto all'elaborazione sequenziale su CPU.