Labeling delle Componenti Connesse su Immagini

Andrea Premate

November 26, 2023

1 Introduzione

L'algoritmo di labeling delle componenti connesse di un'immagine binaria identifica gli insiemi di pixel connessi all'interno dell'immagine, assegnando loro un'etichetta unica. Considerando un'immagine binaria come una griglia di pixel, dove ogni pixel ha un valore binario (0 o 1), l'obiettivo è raggruppare i pixel adiacenti con valore 1 (foreground) in componenti connesse.

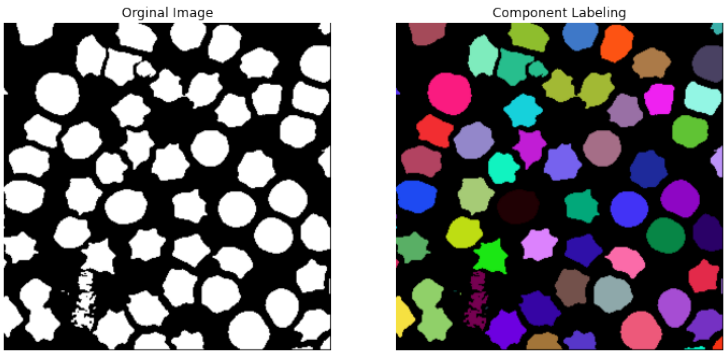

Figure 1: Descrizione grafica di input(sinistra) e output(destra) dell'algoritmo.

L'algoritmo a due passaggi (Two-Pass Algorithm), noto anche come l'algoritmo Hoshen–Kopelman [2], è stato scelto per l'implementazione seriale, MPI e OpenMP.

Per quanto riguarda invece l'implementazione in CUDA, è stato scelto un riadattamento del Two-Pass algorithm, che si basa come quest'ultimo sull'equivalenza di etichette, ma che tiene conto della struttura hardware delle GPU. La sua implementazione è descritta in [3], ispirata da Hawick et al. [4].

Gli algoritmi sono stati testati su immagini binarizzate provenienti dal dataset [1]. In particolare per il controllo qualitativo degli algoritmi sono state utilizzate immagini provenienti dalla sezione "medical", contenenti immagini rappresentanti nuclei di cellule, mentre per il calcolo delle prestazioni sono state utilizzate immagini provenienti dalla sezione "random", contenenti immagini di rumore.

2 Analisi dell'Algoritmo

2.1 Two-Pass Algorithm

L'algoritmo Two-Pass opera in due passaggi sull'immagine: il primo assegna etichette temporanee e registra le equivalenze, mentre il secondo sostituisce ogni etichetta temporanea con l'etichetta più piccola della sua classe di equivalenza.

First Pass:

1. Itera attraverso ogni elemento della matrice, riga per riga (Raster Scanning).
2. Se l'elemento non è parte del background, ottieni gli elementi vicini.
3. Se non ci sono vicini, assegna un'etichetta unica all'elemento corrente e continua.
4. Altrimenti, trova il vicino con l'etichetta più piccola e assegnala all'elemento corrente.
5. Memorizza l'equivalenza tra le etichette dei vicini.

Second Pass:

1. Itera nuovamente attraverso ogni elemento della matrice.
2. Se l'elemento non è sfondo, riassegna l'etichetta con l'etichetta equivalente più bassa.

Le equivalenze delle etichette sono gestite in modo efficiente con una struttura dati Union-Find. Tale struttura dati, nota anche come Disjoint-Set, è utilizzata per tenere traccia di un insieme di elementi partizionati in diversi insiemi disgiunti (non sovrapposti). È particolarmente efficace per gestire l'equivalenza tra elementi in problemi di raggruppamento o clustering. Il suo utilizzo è gestito tramite le funzioni:

- Union: Combina due insiemi in un unico insieme. Se due elementi appartengono a insiemi differenti, l'operazione Union li unisce in un unico insieme.
- Find: Determina a quale insieme appartiene un dato elemento. Questa operazione aiuta a verificare rapidamente se due elementi sono nello stesso insieme.

La struttura di Union-Find è ottimizzata per ridurre la complessità delle operazioni Union e Find.

Parallelizzazione

Per parallelizzare tale algoritmo su p processi, si è quindi divisa l'immagine in p parti, tagliandola orizzontalmente. Successivamente vengono eseguiti entrambi i passaggi in modo autonomo dai processi, producendo così delle equivalenze tra label locali alla sottoporzione di immagine elaborata dal singolo processo. Viene calcolata quindi l'equivalenza tra etichiette in modo globale, andando ad analizzare le righe di confine delle sotto-immagini ed infine viene eseguito nuovamente il second-pass su tutte le sottoimmagini, tenendo conto delle equivalenze a livello globale.

Complessità computazionale

Tale algoritmo ha una complessità temporale e spaziale di $O(N)$, dove N è il numero totale di pixel nell'immagine ($N = W \times H$ per un'immagine larga w pixel e alta h pixel).

- **Inizializzazione**: Calcolare l'altezza e la larghezza dell'immagine è $O(1)$, l'inizializzazione dell'immagine etichettata e dell'oggetto Union-Find è $O(N)$.
- **First Pass**: Per ogni pixel, si esaminano i vicini e si eseguono eventuali operazioni Union-Find. Poiché Union-Find è implementato in modo efficiente, il costo medio è $O(1)$, rendendo il primo passaggio $O(N)$.
- **Second Pass**: Ogni pixel viene controllato e la sua etichetta viene sostituita con l'etichetta rappresentativa della sua classe di equivalenza. Poiché l'operazione Find() ha un costo effettivo di $O(1)$, anche il secondo passaggio è $O(N)$.

L'algoritmo ha quindi una complessità temporale di O(N) e richiede uno spazio aggiuntivo O(N + L), dove L è il numero di etichette create. Tuttavia, poiché il numero massimo di etichette necessarie è N/2 (ad esempio, un modello a scacchiera con 4-connettività), O(N+L) diventa O(N). Pertanto, l'algoritmo richiede O(N) tempo e O(N) spazio.

Per il tempo teorico dell'algoritmo parallelo invece dobbiamo dividere N per il numero p dei processi e poi sommare il tempo di calcolo delle equivalenze a livello globale, che richiede quindi l'analisi di $p-1$ confini, ognuno costituito da 2 righe. Sommiamo poi l'ultimo Second Pass glogale. Il tempo risultante quindi sarà: $O(\frac{N}{p} + (p-1)2W)) + O(N)$ che, considerando $W \approx \sqrt{N}$ e semplificando, risulta essere semplicemente $O(\frac{N}{p})$.

3 Implementazione Seriale

Nello pseudocodice 1 viene presentato l'algoritmo seriale. I tempi di calcolo per l'algoritmo seriale sono illustrati dell'immagine 2 e saranno quelli utilizzati per il calcolo di Speedup ed Efficiency. Tali valori sono riassunti inoltre nella tabella 1.

Algorithm 1 Two-Pass Algorithm Seriale

```
1:  procedure TwoPass(data)
2:      linked ← ∅
3:      labels ← matrice inizializzata a Background
4:      NextLabel ← 0                                          ▷ Primo passaggio
5:      for row in data do
6:          for column in row do
7:              if data[row][column] ≠ Background then
8:                  neighbors ← elementi connessi
9:                  if neighbors = ∅ then
10:                     linked[NextLabel] ← {NextLabel}
11:                     labels[row][column] ← NextLabel
12:                     NextLabel ← NextLabel + 1
13:                 else
14:                     L ← etichette di neighbors
15:                     labels[row][column] ← min(L)
16:                     for label in L do
17:                         linked[label] ← union(linked[label], L)
18:                     end for
19:                 end if
20:             end if
21:         end for
22:     end for                                                ▷ Secondo passaggio
23:     for row in data do
24:         for column in row do
25:             if data[row][column] ≠ Background then
26:                 labels[row][column] ← find(labels[row][column])
27:             end if
28:         end for
29:     end for
30:     return labels
```

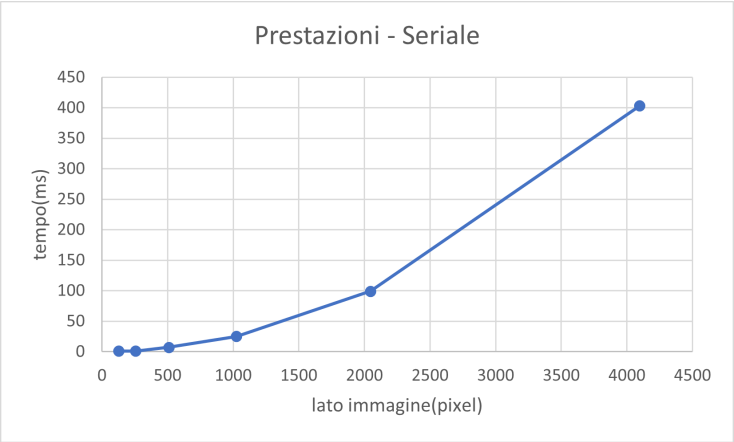

Figure 2: Tempi di calcolo algoritmo seriale.

Lato(pixel)	Tempo(ms)
128	1
256	1
512	7
1024	25
2048	99
4096	403

Table 1: Tempi di esecuzione dell'algoritmo seriale

4 MPI

Nello pseudocodice 2 viene presentato l'algoritmo implementato in MPI. I tempi di calcolo per l'algoritmo MPI sono illustrati dell'immagine 3. Tali valori sono riassunti inoltre nella tabella 2. Nelle immagini 4 e 5 troviamo rispettivamente i grafici che illustrano lo Speedup e l'Efficiency.

Algorithm 2 Two-pass algorithm in MPI

```
procedure TWO-PASS()
    if rank = 0 then
        img ← readimg()
        w ← width(img)
        h ← height(img)
    end if
    MPI_Bcast(h)
    MPI_Bcast(w)
    Calcola rows_per_process
    Alloca recv_buffer
    MPI_Scatterv(img, recv_buffer)
    firstPass(recv_buffer)
    secondPass(recv_buffer)
    if rank = 0 then
        MPI_Gatherv(recv_buffer, img)
        MPI_Gather(localEquivalences)
        globalEquivalences ← findGlobalEquivalences(localEquivalences)
    else
        MPI_Gatherv(recv_buffer, img)
        MPI_Gather(localEquivalences)
    end if
    MPI_Bcast(globalEquivalences)
    secondPass(recv_buffer)
    MPI_Gatherv(recv_buffer, img)
end procedure=0
```

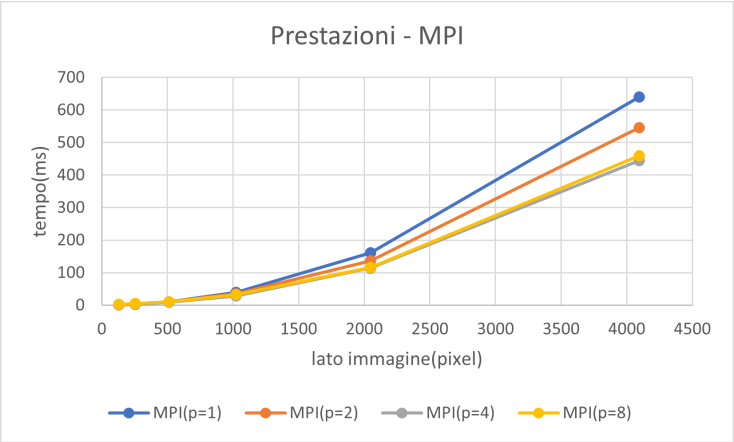

Figure 3: Tempi di calcolo algoritmo MPI.

Lato(pixel)	p=1(ms)	p=2(ms)	p=4(ms)	p=8(ms)
128	1	1	2	3
256	3	3	3	5
512	10	10	9	11
1024	40	35	29	33
2048	161	137	114	116
4096	640	545	445	460

Table 2: Tempi di esecuzione dell'algoritmo in base alla dimensione dell'immagine e al numero di processi MPI

Andando ad analizzare lo Speedup, sempre minore di 1, si può quindi constatare che l'utilizzo di MPI non sembra essere utile al velocizzare il task di labeling delle componenti connesse. Ciò è probabilmente dovuto al fatto che l'invio tra processi di porzioni di immagini e relativa struttura dati Union-Find introduce troppo overhead. Non si ottiene così un vantaggio in termini di tempi.

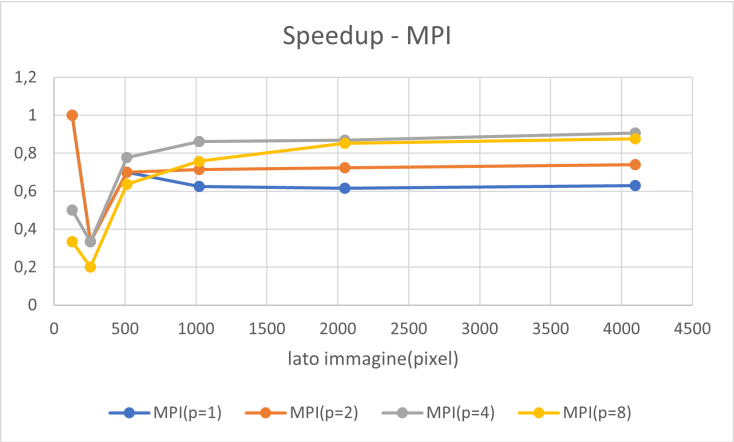

Figure 4: Speedup algoritmo MPI.

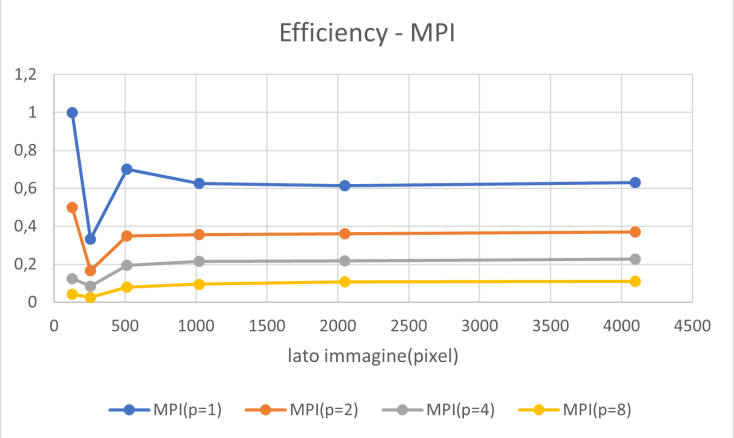

Figure 5: Efficiency algoritmo MPI.

5 OpenMP

Nello pseudocodice 3 viene presentato l'algoritmo implementato in MPI. I tempi di calcolo per l'algoritmo OMP sono illustrati dell'immagine 6. Tali valori sono riassunti inoltre nella tabella 3. Nelle immagini 7 e 8 troviamo rispettivamente i grafici che illustrano lo Speedup e l'Efficiency.

Algorithm 3 Two-pass algorithm in omp

```
procedure TWO-PASS()
    img ← readimg()
    w ← width(img)
    h ← height(img)
    rows_per_thread ← h / n_threads
    extra_rows ← h % n_threads
    #pragma omp parallel num_threads(n_threads)
        thread_id ← omp_get_thread_num()
        Calcola start_row e end_row
        firstPass(img, start_row, end_row))
        secondPass(img, start_row, end_row)
        findGlobalEquivalences(img, start_row)
        secondPass(img, start_row, end_row)
end procedure
```

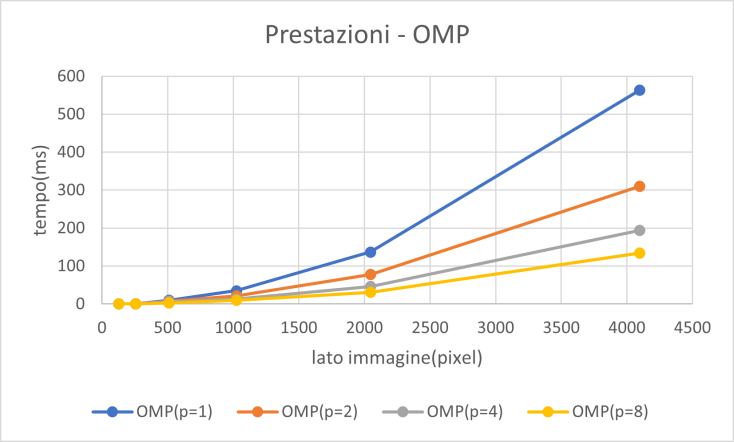

Figure 6: Tempi di calcolo algoritmo OMP.

Lato(pixel)	p=1(ms)	p=2(ms)	p=4(ms)	p=8(ms)
128	1	1	1	1
256	1	1	1	1
512	10	7	6	3
1024	35	21	13	10
2048	137	78	46	31
4096	563	310	194	134

Table 3: Tempi di esecuzione dell'algoritmo in base alla dimensione dell'immagine e al numero di processi OMP

In questo caso si ha uno Speedup maggiore di 1 quando si utilizzano 2 o più processi. In particcolare all'aumentare di p lo Speedup aumenta fino ad ottenere per $p = 8$ uno Speedup di 3.

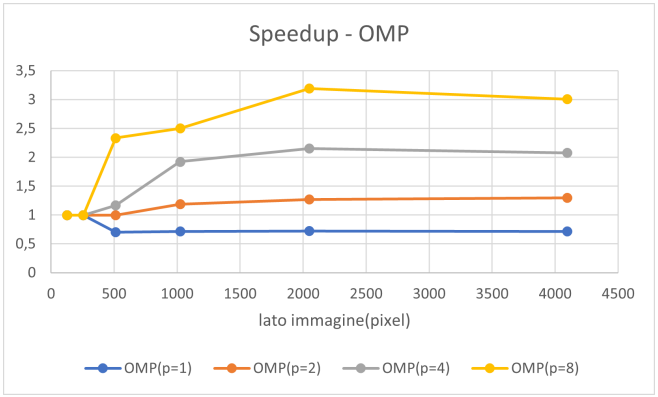

Figure 7: Speedup algoritmo OMP.

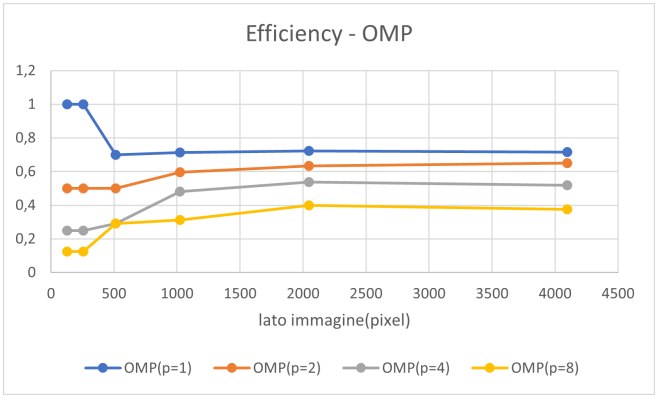

Figure 8: Efficiency algoritmo OMP.

6 CUDA

In questo caso, come anticipato nell'introduzione, si utilizza una variante dell'algoritmo two-pass. L'algoritmo è iterativo e comprende tre fasi:

1. Inizializzazione: ad ogni pixel viene assegnata una label differente, coincidente con la propria posizione all'interno della matrice(Raster Scanning). (Pseudocodice 4).

2. Loop Scanning-Analysis finché ci sono elementi da aggiornare

   (a) Scanning: per ogni pixel in posizione i con valore p, guarda i 4 pixel vicini e se ne trovi uno minore di p, assegna al pixel in posizione p quel valore (se minore del valore in posizione p) e setta la flag IsNotDone. (Pseudocodice 5).

   (b) Analysis: è la fase di rietichettatura. Qui all'interno di un ciclo while, ogni thread esplora una sequenza di etichette a partire dalla posizione corrente, considerandole come riferimenti. Il ciclo termina quando viene trovata l'etichetta il cui valore coincide con l'indice di quell'elemento. Quindi viene assegnata alla posizione corrente quell'etichetta. (Pseudocodice 6). Nella figura 9 è raffigurato il processo di Analysis partendo dalla posizione 11 ed esplorando le etichette seguendo i riferimenti, fino ad arrivare in posizione 1, valore che sarà assegnato alla posizione 11.

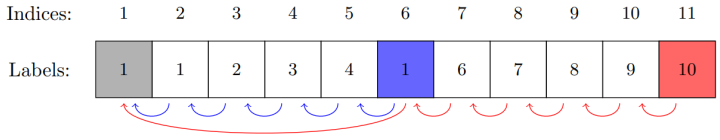

Figure 9: Procedura di Analysis [3].

Lo pseudocodice dell'host invece è indicato nell'algoritmo 7.

Si noti che non è mai necessario utilizzare operazioni atomiche o sincronizzazioni, che rallenterebbero l'esecuzione, a causa della natura iterativa dell'algoritmo: se avvengono delle collisioni verranno risolte al successivo step di iterazione.

I tempi di calcolo per l'algoritmo CUDA sono illustrati dell'immagine 10. Tali valori sono riassunti inoltre nella tabella 4. Nell' immagine 11 troviamo il grafico relativo allo Speedup. Non è possibile calcolare facilmente l'Efficiency poichè richiede una comprensione dettagliata del numero di Streaming Multiprocessors (SMs) allocati e della occupancy dei thread.

Algorithm 4 InitLabels

```
1: procedure InitLabels(Labels)
2:     id ← blockIdx.y × gridDim.x × blockDim.x + blockIdx.x × blockDim.x
           + threadIdx.x
3:     cy ← id / SIZEX
4:     cx ← id - cy × SIZEX
5:     aPos ← (cy + 1) × SIZEXPAD + cx + 1
6:     l ← Labels[aPos]
7:     l ← l × aPos
8:     Labels[aPos] ← l
9: end procedure
```

Algorithm 5 Scanning

```
1: procedure Scanning(Labels, IsNotDone)
2:     id ← blockIdx.y × gridDim.x × blockDim.x + blockIdx.x × blockDim.x
           + threadIdx.x
3:     cy ← id / SIZEX
4:     cx ← id % SIZEX
5:     aPos ← (cy + 1) × SIZEXPAD + cx + 1
6:     if aPos < SIZEXPAD × SIZEXPAD then
7:         l ← Labels[aPos]
8:         if l > 0 then
9:             lw ← Labels[aPos - 1]
10:            le ← Labels[aPos + 1]
11:            ls ← Labels[aPos - SIZEX - 2]
12:            ln ← Labels[aPos + SIZEX + 2]
13:            minl ← min(lw, le, ls, ln non di background)
14:            if minl < l then
15:                ll ← Labels[l]
16:                Labels[l] ← min(ll, minl)
17:                IsNotDone[0] ← 1
18:            end if
19:        end if
20:    end if
21: end procedure
```

Algorithm 6 Analysis

```
1: procedure Analysis(Labels)
2:     id ← blockIdx.y × gridDim.x × blockDim.x + blockIdx.x × blockDim.x
           + threadIdx.x
3:     cy ← id / SIZEX
4:     cx ← id % SIZEX
5:     aPos ← (cy + 1) × SIZEXPAD + cx + 1
6:     if aPos < SIZEXPAD × SIZEXPAD then
7:         label ← Labels[aPos]
8:         if label > 0 then
9:             r ← Labels[label]
10:            while r ≠ label do
11:                label ← Labels[r]
12:                r ← Labels[label]
13:            end while
14:            Labels[aPos] ← label
15:        end if
16:    end if
17: end procedure
```

Algorithm 7 Codice host

```
1: procedure HostCUDA
2:     h_img_nopadding ← readimg()
3:     w ← width(h_img_nopadding)
4:     h ← height(h_img_nopadding)
5:     h_img ← add_padding(h_img_no_padding)
6:     cudaMalloc(d_img)
7:     cudaMemcpy(d_img, h_img)                          ▷ HostToDevice
8:     threadsPerBlock ← dim3(th_b)
9:     blocksPerGrid ← dim3((w + 2 + threadsPerBlock.x - 1) /
       threadsPerBlock.x, (h+2 + threadsPerBlock.y - 1) / threadsPerBlock.y)
10:    cudaMalloc(d_isNotDone)
11:    cudaMemcpy(d_isNotDone, h_isNotDone)              ▷ HostToDevice
12:    InitLabels<<<blocksPerGrid, threadsPerBlock>>>(d_img, w)
13:    repeat
14:        h_isNotDone ← 0
15:        cudaMemcpy(d_isNotDone, h_isNotDone)          ▷ HostToDevice
16:        Scanning<<<blocksPerGrid, threadsPerBlock>>>(d_img,d_isNotDone,
           w, h)
17:        Analysis<<<blocksPerGrid, threadsPerBlock>>>(d_img, w, h)
18:        cudaMemcpy(h_isNotDone, d_isNotDone)          ▷ DeviceToHost
19:    until h_isNotDone = 0
20:    cudaMemcpy(h_img, d_img)                          ▷ DeviceToHost
21:    cudaFree(d_img)
22: end procedure
```

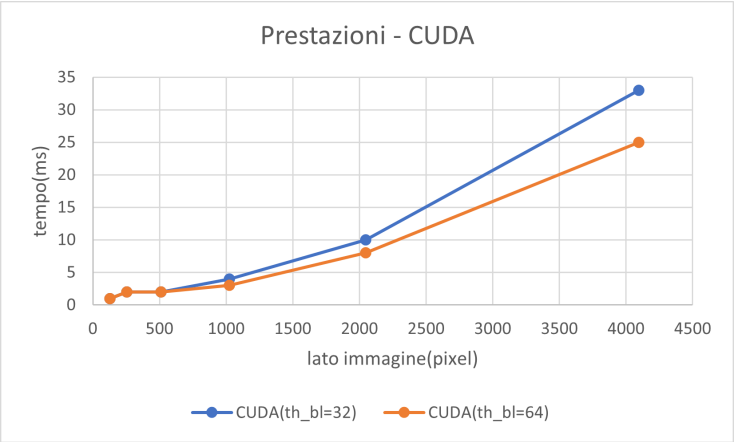

Figure 10: Tempi di calcolo algoritmo CUDA.

Lato(pixel)	th_bl=32(ms)	th_bl=64(ms)
128	1	1
256	2	2
512	2	2
1024	4	3
2048	10	8
4096	33	25

Table 4: Tempi di esecuzione dell'algoritmo CUDA in base alla dimensione dell'immagine e al numero di thread per blocco

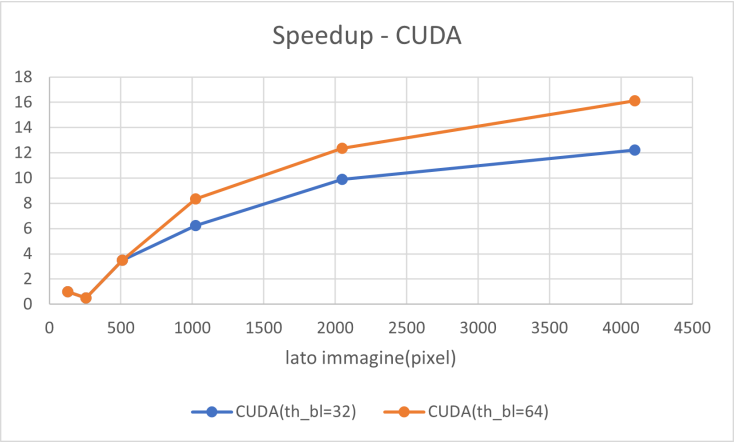

Figure 11: Speedup algoritmo CUDA.

7 Risultati e Confronti

Nella Figura 12, si osserva un confronto tra le configurazioni ottimali per ciascuna delle implementazioni analizzate, in termini di numero di processi o thread per blocco. È evidente che l'implementazione MPI sia meno performante rispetto a quella seriale. Invece, le implementazioni OpenMP e, soprattutto, CUDA emergono come le più efficienti. Questo vantaggio è attribuibile all'utilizzo della memoria condivisa in queste ultime due, che consente di evitare il significativo sovraccarico associato al trasferimento di grandi strutture dati, una problematica comune nell'approccio MPI.

Questa differenza di prestazioni è tangibile anche per immagini con dimensioni "comuni", ad esempio con lato da 1000 pixel, mentre inizia ad essere trascurabile solo per immagini molto piccole.

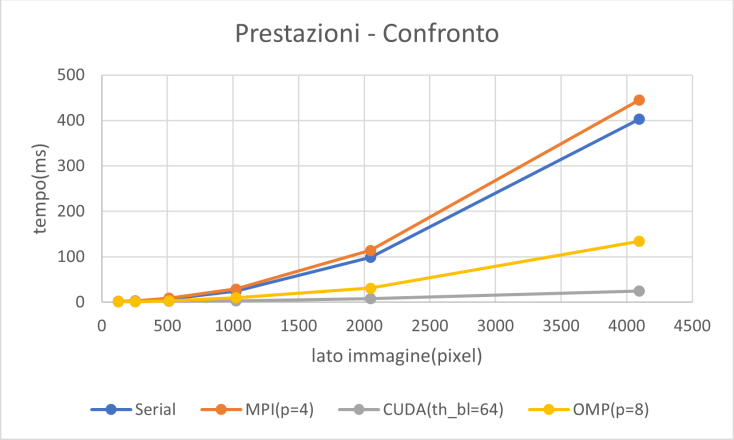

Figure 12: Tempi di calcolo a confronto.

References

[1] Costantino Grana et al. "YACCLAB - Yet Another Connected Components Labeling Benchmark". In: *2016 23rd International Conference on Pattern Recognition (ICPR)*. 2016, pp. 3109–3114. DOI: 10.1109/ICPR.2016.7900112.

[2] J. Hoshen and R. Kopelman. "Percolation and cluster distribution. I. Cluster multiple labeling technique and critical concentration algorithm". In: 14.8 (Oct. 1976), pp. 3438–3445. DOI: 10.1103/PhysRevB.14.3438.

[3] Oleksandr Kalentev et al. "Connected component labeling on a 2D grid using CUDA". In: *J. Parallel Distributed Comput.* 71 (2011), pp. 615–620. URL: https://api.semanticscholar.org/CorpusID:18940008.

[4] Daniel Peter Playne and Ken Hawick. "A New Algorithm for Parallel Connected-Component Labelling on GPUs". In: *IEEE Transactions on Parallel and Distributed Systems* 29.6 (2018), pp. 1217–1230. DOI: 10.1109/TPDS.2018.2799216.