Parallel Computing Final project: Local Binary Pattern

Marco Solarino
E-mail address
marco.solarino@stud.unifi.it

Simone Pezzulla
E-mail address
simone.pezzulla@stud.unifi.it

Abstract

In questa relazione si tratta del progetto finale di Parallel Computing in cui si implementa l'algoritmo Local Binary Pattern per immagini. Si discuterà delle sue tre versioni: sequenziale, CUDA e OpenMP e in particolare se ne confronteranno e commenteranno le prestazioni in termini di tempo di esecuzione e speedup.

1. Introduzione

L'LBP (Local Binary Pattern) è un algoritmo nato per problemi di classificazione nell'ambito della Computer Vision (e.g. riconoscimento facciale). Data un'immagine, per ogni pixel si confronta il suo livello di grigio con quello dei suoi 8 vicini in senso orario e si costruisce un vettore delle feature. Se il vicino ha un livello di grigio più alto del pixel centrale si inserisce nel vettore un 1, viceversa uno 0. Il vettore rappresenta un numero binario a 8 cifre, il corrispondente decimale sarà il nuovo valore del pixel in esame. Un volta terminato il processo per l'intera immagine è possibile calcolarne l'istogramma e usarlo per algoritmi di classificazione.

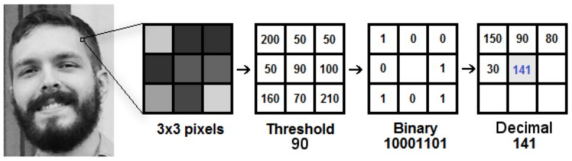

Figura 1: Caption

1.1. Linguaggi e framework

Sono state implementate tre versioni dell'algoritmo:

- **Sequenziale**: la versione usata per confrontare le prestazioni con le versioni parallele, scritta in C++17.
  Codice disponibile qui: `https://github.com/MarcoSolarino/LBPSequential`
- **CUDA**: la versione parallela su GPU scritta in C/C++ usando Nsight su sistema Linux.
  Codice disponibile qui: `https://github.com/daikon899/LBP_CUDA`
- **OpenMp**: versione parallela su CPU, scritta in C++17.
  Codice disponibile qui: `https://github.com/daikon899/LBP_OpenMP`

1.2. Specifiche delle macchine

Di seguito le specifiche delle macchine su cui sono stati eseguiti i test sulle prestazioni:

- **Workstation di Santa Marta**:
  Per la versione CUDA. Le specifiche sono le seguenti.
  - processore Intel® Core™ i7-860
  - scheda grafica NVIDIA GeForce GTX 980 Ti
  - sistema operativo ubuntu Xfce
- **MacBook Pro late 2016**:
  Per la versione OpenMp:
  - processore Intel Core i5 dual core (quattro core logici)
  - 8GB RAM
  - macOS Catalina

2. CUDA

La versione CUDA è costituita da un solo kernel che calcola per ogni pixel il vettore e dunque il nuovo valore e inoltre si occupa di aggiornare l'istogramma.
Sono state implementate due versioni del kernel:

- **lbpApply**: si effettuano gli accessi direttamente in global memory.
- **lbpApplyV2**: si fa uso della shared memory. Ogni blocco salva localmente una porzione di immagine che ha dimensione pari a quella del blocco di thread più

la cornice dei pixel di bordo. Allo stesso modo l'istogramma viene prima salvato localmente e una volta che tutti i thread del blocco hanno completato questa operazione (garantito dalla barriera di sincronizzazione *__syncthreads*) l'istogramma in global memory viene aggiornato tramite gli *atomicAdd*.

2.1. Test e risultati

2.1.1 Test al variare della dimensione dell'immagine

È stato misurato il tempo di esecuzione della versione sequenziale e delle due versioni CUDA (con e senza l'uso di shared memory) partendo da un'immagine di dimensione 200x200 e moltiplicando la dimensione fino a 40 volte. I risultati sono mostrati nelle figure 2 e 3.

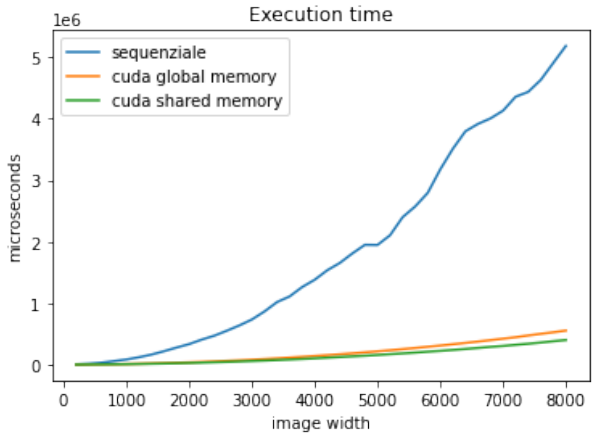

Figura 2: Comparazione tempi di esecuzione nelle tre versioni

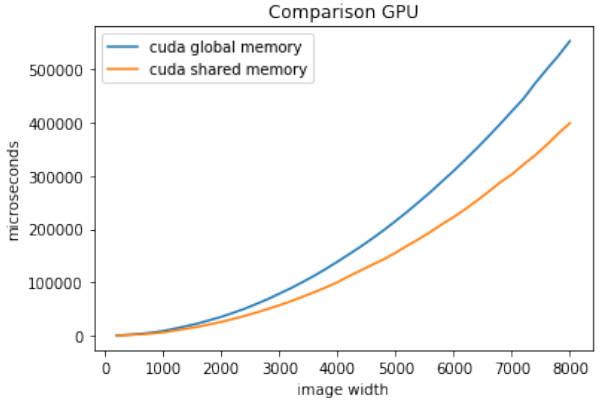

Figura 3: Comparazione tempi di esecuzione delle versioni CUDA in shared e global memory

Il tempo di esecuzione della versione sequenziale supera di molto quello della versione CUDA, mentre confrontando le due versioni di quest'ultima (figura 3) si nota come l'utilizzo di shared memory apporti migliori performance poichè il valore di un pixel viene letto più volte (fino a un massimo di 9) e quindi il numero di accessi in memoria globale della prima versione è elevatissimo. Lo stesso discorso vale per l'istogramma.

Dimensione	Sequenziale	CUDA
20x20	258 μs	563 μs
40x40	285 μs	490 μs
60x60	391 μs	461 μs
80x80	662 μs	553 μs
100x100	1369 μs	477 μs

Tabella 1: Confronto dei tempi della versione sequenziale e CUDA

Nello stesso test è stato valutato lo speedup. I risultati sono mostrati in figura 4 e mettono a confronto le due versioni del kernel.

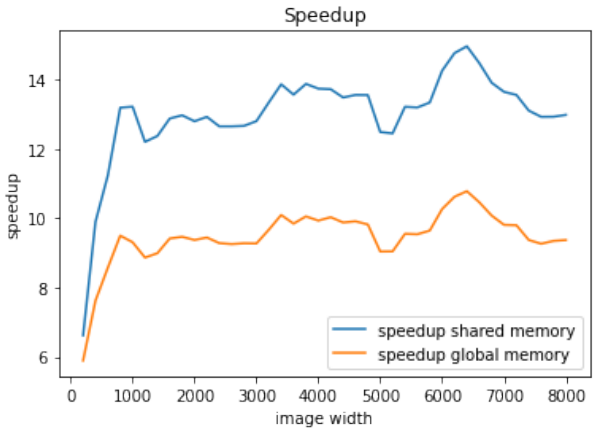

Figura 4: Speedup al variare della dimensione dell'immagine

I due andamenti sono similari: aumentano rapidamente per piccole dimensioni dell'immagine e si stabilizzano asintoticamente a dimensioni maggiori. E' chiaro come usando la shared memory si ottenga uno speedup maggiore (fino a 14 volte più veloce della versione sequenziale).

In ultima analisi, sono stati studiati i tempi di esecuzione per immagini molto piccole, per verificare come in questi casi la parallelizzazione perde di significato e presenta prestazioni peggiori della versione sequenziale a causa dei costi di overhead che rappresentano uno svantaggio troppo grande a fronte del piccolo numero di dati a disposizione. I risultati sono mostrati in figura 5.

Come mostrato anche nella tabella 1 la versione sequenziale ha migliori prestazioni per immagini che hanno una

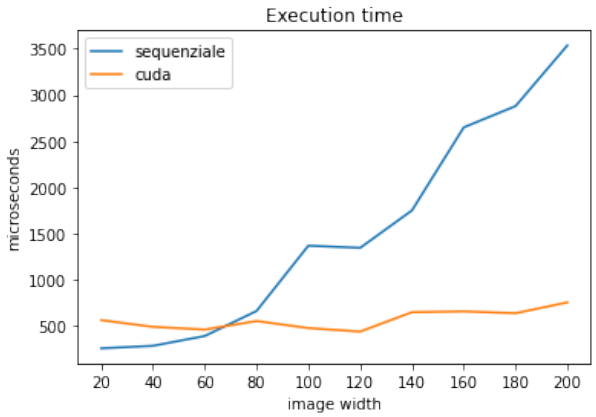

Figura 5: Tempi di esecuzione per immagini molto piccole

dimensione fino a 60x60, dopodiché la situazione si inverte.

2.1.2 Test al variare della dimensione del blocco

Come ultimo test è stato misurato il tempo di esecuzione al variare della dimensione del blocco dei thread con un'immagine 4k. I risultati sono mostrati in figura 6.

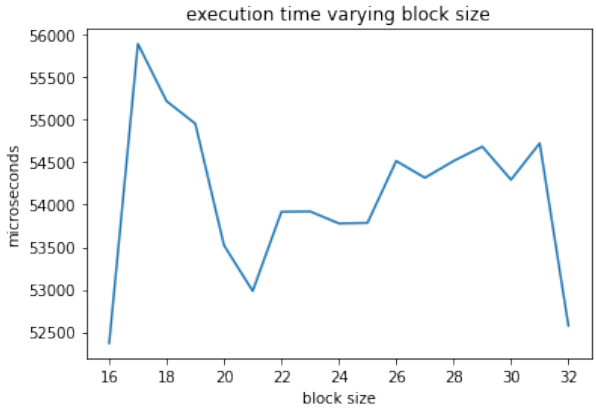

Figura 6: tempi di esecuzione al variare della dimensione del blocco dei thread. I blocchi sono quadrati.

Si nota come la dimensione migliore per performance sia la 16x16, ovvero per blocchi di 256 thread che è un multiplo di 32 (la dimensione di un warp). Per le dimensioni non multiple di 32 infatti, si hanno core inutilizzati e prestazioni peggiori. Inoltre nel caso dei blocchi 16x16 si riesce a massimizzare il numero dei thread per *Streaming Multiprocessor*, ciò risulta in un *latency hiding* migliore e dunque in prestazioni migliori. Le stesse considerazioni valgono per blocchi di dimensione 32x32 (1024 thread) dove si riscontrano tempi di esecuzione simili al caso 16x16. Anche in questo caso si massimizzano i thread nello SM.

2.1.3 Nota sui test

Ogni dato è stato ottenuto facendo la media su 50 iterazioni.

3. OpenMP

La versione OpenMP non ha subito modifiche alla struttura rispetto alla versione sequenziale: due cicli for annidati si occupano di calcolare i vettori delle feature per ogni pixel. La maggior parte delle variabili e array sono state condivise tra i thread tramite la direttiva *shared* (il vettore dei pesi e le immagini in input e in output). Per l'istogramma invece i thread effettuano un'operazione di riduzione (addizione) al termine dell'esecuzione.

3.1. Test e risultati

3.1.1 Test al variare della dimensione dell'immagine

Come nel caso della versione CUDA, è stato misurato e messo a confronto con la versione sequenziale il tempo di esecuzione al variare della dimensione dell'immagine. L'immagine di partenza ha ancora una volta dimensione 200x200, che è stata aumentata gradualmente fino a 40 volte. Inoltre è stata confrontata la differenza di performance con l'uso o meno di *collapse(2)* per i cicli annidati.

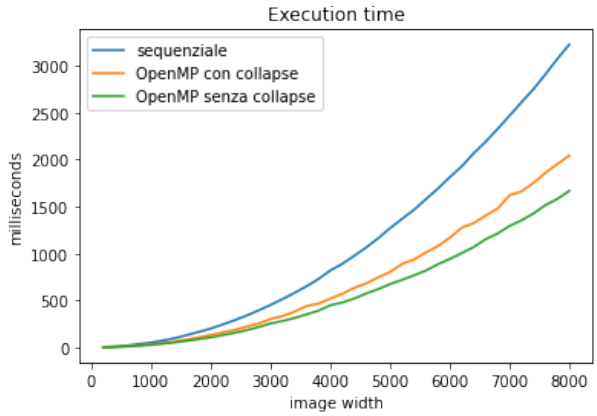

Figura 7: tempi di esecuzione al variare della dimensione dell'immagine

Come mostrato in figura 7 e 8 ancora una volta si ha un miglioramento delle prestazioni, anche se di gran lunga inferiore a quello riscontrato con l'utilizzo della GPU. Inoltre la versione che fa uso di *collapse* risulta avere prestazioni peggiori, ciò deriva dal fatto che i cicli sono bilanciati.

Interessante notare come ancora una volta, per dimensioni piccole delle immagini la parallelizzazione perde di senso per via dei costi di overhead. (fig. 9).

L'immagine di partenza in questo caso ha dimensione 20x20 che è stata incrementata fino a 200x200. La versione sequenziale ha prestazioni migliori per le immagini

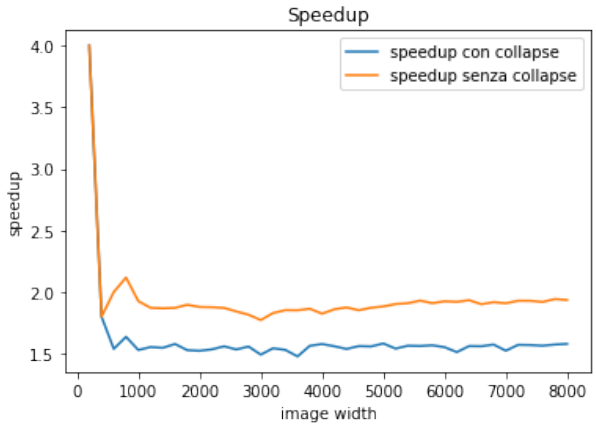

Figura 8: valutazione dello speedup al variare della dimensione dell'immagine

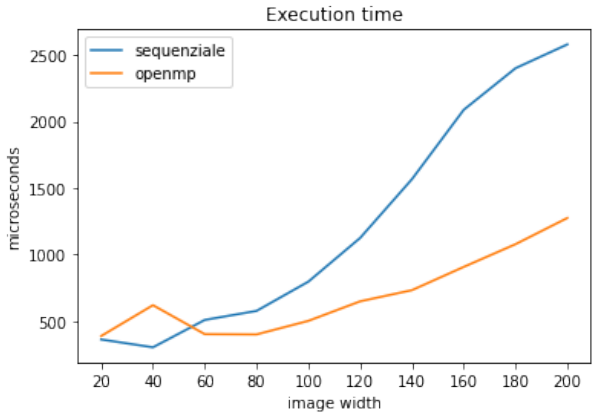

Figura 9: tempo di esecuzione al variare della dimensione dell'immagine (per piccole dimensioni)

Dimensione	sequenziale	OpenMP
20x20	361 μs	387 μs
40x40	302 μs	619 μs
60x60	508 μs	401 μs
80x80	576 μs	399 μs
100x100	796 μs	501 μs

Tabella 2: Confronto dei tempi della versione sequenziale e OpneMP

di dimensione fino a 40x40 per poi essere superato dalla versione OpenMP come mostrato anche nella tabella 2.

3.1.2 Test al variare del numero dei thread

E' stato valutato infine il tempo di esecuzione al variare del numero dei thread. Le prestazioni migliorano fino a che i thread raggiungono il numero di core logici per poi rimanere all'incirca costanti. I risultati sono mostrati in figura 10 e nella tabella 3.

# thread	OpenMP
1	651 μs
2	350 μs
3	311 μs
4	273 μs
5	277 μs

Tabella 3: Confronto dei tempi di esecuzione all'aumentare del numero di thread

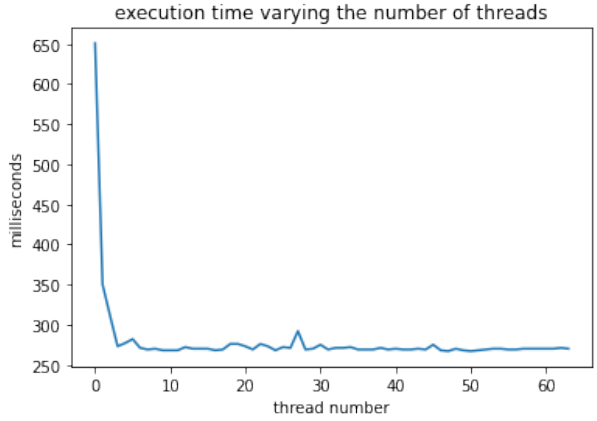

Figura 10: tempo di esecuzione al variare del numero dei thread

3.1.3 Nota sui test

Ogni dato dei test è stato ottenuto, come nel caso della versione CUDA, facendo la media su 50 iterazioni.

4. Conclusioni

I test su entrambe le versioni hanno reso evidente come l'uso della parallelizzazione sia, nel caso in cui vi sono abbastanza dati da processare, il modo più efficace di ottenere delle buone prestazioni. CUDA si è dimostrato essere, tra le due soluzioni apportate, la migliore, per via delle maggiori risorse della GPU e poichè il problema preso in esame non richiedeva particolare sincronizzazione tra i thread.