Progetto di Gestione Dell'Informazione Geospaziale - DBSCAN

Damiano Bianda

22 dicembre 2018

Sommario

In questo progetto si vuole implementare l'algoritmo di clustering DBSCAN tramite Java.
Dato un insieme di coordinate proiettate, queste vengono classificate secondo il cluster d'appartenenza o come outliers.
Infine i risultati ottenuti sono rappresentati attraverso una mappa utilizzando il software QGIS.

1 Algoritmo

1.1 Introduzione

DBSCAN è un algoritmo di clustering partitivo basato sulla densità. I principali vantaggi sono che è possibile scoprire cluster di forme arbitraria (dimensioni e forme differenti) e identificare gli outliers, ossia i punti che sono in un'area densamente popolata e quindi non appartengono a nessun cluster. I svantaggi sono invece che essendo un algoritmo parametrico è necessario definire dei parametri in base al tipo di dati da analizzare e che alcuni dataset presentano il problema della densità variabile, ossia sono presenti più cluster ma a densità diverse e quindi dati dei parametri non è possibile in un esecuzione identificarli tutti.

1.2 Definizioni

DBSCAN è parametrizzato tramite ε e MinPoints.
Il vicinato di un punto sono tutti i punti che ricadono nel cerchio con centro pari alla sua coordinata e raggio ε.
Un punto è detto:

- core point se il suo vicinato contiene almeno MinPoints elementi.
- border point se non è un core point, ma è nel vicinato di uno o più core point
- noise point se non è nè core point, nè border point

Le definizioni seguenti descrivono un rapporto di connessione tra i punti e servono per definire il concetto di cluster:

- directly density-reachable
  un punto p è detto directly density-reachable da un punto q se p è nel vicinato di q e se q è un core point
- density-reachable
  un punto p è detto density-reachable da un punto q se c'è una serie di punti, in cui il primo è q e l'ultimo è p, dove ogni elemento è directly density-reachable dal precedente
- density-connected
  un punto p è detto density-connected ad un punto q se c'è un punto r tale che p e q sono density-reachable da r

Quindi un cluster è un insieme massimo di punti density-connected.

1.3 Esecuzione di DBSCAN

DBSCAN itera su tutti i punti presenti nel dataset e per ognuno si determina se è un core point.
In caso positivo si crea un cluster che viene espanso coi punti presenti nel suo vicinato.
Il processo si ripete iterativamente controllando i punti appena aggiunti, se a loro volta sono core point viene aggiunto il loro vicinato e così via fin quando tutti i punti density-reachable sono stati inglobati nel cluster.
Il vicinato non viene aggiunto al cluster quando un punto è border point, poichè non esistono punti density-reachable da questo.
Una volta terminato l'algoritmo si ottiene un dataset con i suoi elementi etichettati secondo i differenti cluster o come noise.

2 Implementazione

2.1 Introduzione

Si è deciso di sfruttare la capacità di parallelismo messa a disposizione dalla GPU facendo in modo che ogni thread calcoli esattamente un LBP.
Sono stati implementati due kernel differenti, il primo che utilizza la shared memory ed il secondo che utilizza la texture memory in modo da potere confrontare le implementazioni (LocalBinaryPatternGPUShared.cu e LocalBinaryPatternGPUTexture.cu).
Molte funzioni __device__ tra i due file sono le stesse, questo per motivi di performance, infatti chiamare funzioni __device__ da un altro file richiede di compilare il codice in modo relocatable (-rdc=true), generando però tempi d'esecuzione maggiori.
Di seguito viene spiegato com'è stato implementato l'algoritmo, facendo riferimento al codice coinvolto alla fine di ogni sezione.

2.1.1 Dimensione della griglia e dei blocchi - lato host

Lavorando su una matrice di valori d'intensità è naturale utilizzare una griglia 2D e blocchi di threads 2D.
La dimensione di blocco, è quella che empiricamente ha dato i tempi migliori, è fissata a 16x16.
La dimensione della griglia viene calcolata a runtime, facendo in modo che ogni pixel dell'immagine abbia un thread associato.

Utils.h:

- const dim3 BLOCK_SIZE(16, 16)
- dim3 getGridSize(GrayScaleImage grayScaleImage)

2.2 Implementazione tramite shared memory

2.2.1 Utilizzo della constant memory - lato host

Ogni thread deve ottenere i valori d'intensità dei *points* punti sulla circonferenza di raggio *radius* per poi costruire il LBP.
I valori vengono calcolati tramite interpolazione bilineare, quindi per il calcolo dell'intensità di un punto il thread necessita di:

1. la coordinata relativa di uno dei quattro pixel coinvolti nel calcolo dell'interpolazione bilineare, precisamente quello in alto a sinistra, le altre tre vengono ricavate da questa.
   (relativa rispetto pixel centrale su cui si calcola il LBP, non a p)
2. la distanza α in direzione x tra il punto ed i pixel alla sua sinistra
3. la distanza β in direzione y tra il punto ed i pixel sovrastanti

la seguente funzione restituisce il valore interpolato:

$$p = tex(x,y)(1-\alpha)(1-\beta)+tex(x+1,y)\alpha(1-\beta)+tex(x,y+1)(1-\alpha)\beta+tex(x+1,y+1)\alpha\beta \quad (1)$$

questi valori sono costanti durante tutta l'esecuzione, quindi sono stati precalcolati e salvati in constant memory in modo da essere acceduti simultaneamente da più threads il più velocemente possibile, evitando calcoli onerosi e ripetuti nel kernel (il calcolo coinvolge le funzioni round, abs, sin e cos).

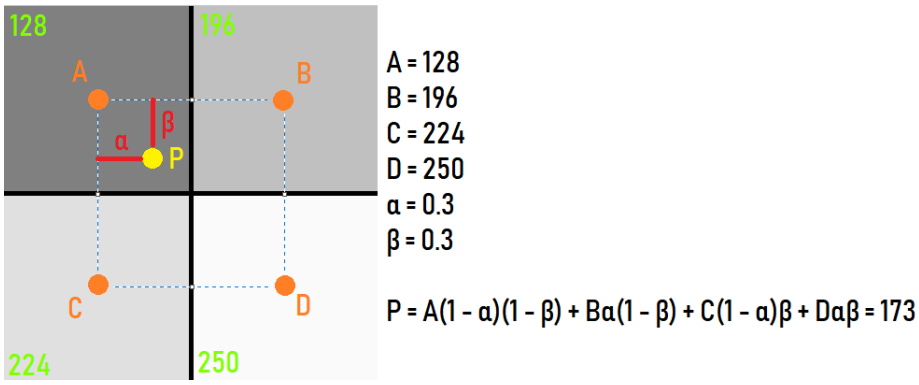


Figura 1: Esempio di interpolazione bilineare

Point.h:

- void surroundingBilinearPoints(const int points, const float radius, Point** bilinearNeighborhoodCoordinates, PointF** bilinearParameters);

LocalBinaryPatternGPUShared.cu:

- __constant__ static Point neighboroodRelativeCoordinates[MAX_POINTS]
- __constant__ static PointF bilinearFilterParameters[MAX_POINTS]
- void copyNeighborhoodCoordsInConstantMemory(Point* points, int length)
- void copyBilinearParamsInConstantMemory(PointF * points, int length)
- Histogram* LocalBinaryPatternGPUSharedMemory(GrayScaleImage* grayScaleImage, int points, float radius, bool bilinear, double* time)

2.2.2 Utilizzo template metaprogramming - lato host

L'algoritmo lato device prevede che un thread iteri sui *points* punti presenti sulla circonferenza di raggio *radius*, utilizzando direttamente la direttiva #pragma unroll *points* per fare l'unloop non portava inizialmente benefici in quanto il valore *points* non è conosciuto a tempo di compilazione.
Tramite l'utilizzo dei templates, della ricorsione e sapendo che la variabile *points* è limitata tra 1 e 32 è possibile emulare un ciclo for lato host, in cui viene invocato il kernel con valore di *points* crescente ad ogni iterazione ed il compilatore è in grado di compilare le diverse versioni del kernel e fare eventuali ottimizzazioni.
Utilizzando questa tecnica il tempo d'esecuzione ha avuto un buon miglioramento a scapito di un eseguibile di grandezza maggiore, ma comunque irrisoria.

LocalBinaryPatternGPUShared.cu:

- template<int staticIteration>
  void selectKernel(int dynamicIteration, bool bilinear, int sharedSize, GrayScaleImage d_image, Histogram d_histogram, unsigned int mask, int borderLength)

2.2.3 Utilizzo della shared memory - lato device

Ogni thread appartenente ad un blocco deve accedere al pixel su cui calcola il LBP e ai pixels circostanti la cui distanza massima in una delle due dimensioni è *borderLength*.
borderLength viene calcolato prima di lanciare il kernel iterando sulle coordinate relative dei pixel top-left e cercando il valore massimo in una delle dimensioni x e y, questo valore viene incrementato di 1 poichè durante l'interpolazione è necessario accedere ai pixels adiacenti e sottostanti a quello top-left a distanza 1.
Gli accessi alla memoria possono essere velocizzati con la lettura da shared memory, quindi un blocco 2D di threads deve caricare in shared memory una porzione dell'immagine pari a:

$$(blockDim.x + 2\ borderLength) \quad x \quad (blockDim.y + 2\ borderLength) \qquad (2)$$

Tramite un doppio ciclo for si incrementano due offset, uno in direzione x e l'altro in y, che vengono aggiunti alle coordinate di ogni thread del blocco.
Un offset parte da un valore di (-*borderLength*) ed incrementato ad ogni iterazione di blockDim._ fin quando raggiunge (blockDim._ + *borderLength*), ossia il valore che permette a tutte le celle di shared memory di essere scritte da un thread del blocco.
Ad ogni iterazione i threads aggiungono gli offset alle proprie coordinate in modo da leggere il valore corretto da global memory e scriverlo in shared memory, controllando che la coordinata di scrittura sia nei limiti della shared memory e scrivendo 0 in caso la coordinata di lettura sia fuori dai limiti dell'immagine (0-padding).
Al termine di questa operazione preliminare i threads si coordinano ed è possibile cominciare il calcolo del LBP.

LocalBinaryPatternGPUShared.cu:

- __device__ static void copyImageToSharedMemory(GrayScaleImage grayScaleImage, unsigned char* sharedTile, int borderLength, int sharedWidth, int sharedHeight)

2.2.4 Calcolo del LBP - lato device

Il LBP generato da un thread è un vettore binario contenuto in una variabile a 32 bit.
Un thread itera *points* volte, ad ogni iterazione legge dalla constant memory le distanze α , β e la coordinata del pixel top-left coinvolto nel calcolo dell'interpolazione, le altre essendo adiacenti vengono ricavate, tramite questi valori si

recuperano le intensità dalla shared memory e si calcola l'intensità del punto p corrente.
Successivamente l'intensità viene confrontata con quella del pixel centrale e si sovrascrive con 1 o 0 l'ultimo bit della variabile LBP, precedentemente shiftata verso sinistra di una posizione. L'operazione di confronto viene effettuata tramite operatori bitwise in modo da evitare eventuali divergenze all'interno di una warp, sfruttando il fatto che il confronto con >= restituisce 1 o 0, ossia il valore che viene scritto.

LocalBinaryPatternGPUShared.cu:

- template<int points>
  __device__ static unsigned int calculateLocalBinaryPatternBilinear(int borderLength, unsigned char* sharedMemory, int sharedWidth, int sharedHeight)

2.2.5 Conteggio delle transizioni del LBP - lato device

Per contare il numero di transizioni presenti in un LBP (01 e 10) si sono utilizzati gli operatori bitwise, in modo da ottenere il conteggio con un numero fisso di istruzioni indipendentemente dalla lunghezza del vettore binario, evitando di iterare sui suoi bit.
Facendo la xor tra il vettore binario LBP e la sua versione ruotata di una posizione si ottiene una parola binaria contenente un numero di 1 pari al conteggio delle transizioni, l'intrinsic __popc(unsigned int x) di CUDA permette di contarli con un unica istruzione.

LocalBinaryPatternGPUShared.cu:

- __device__ static unsigned int countBitSwitches(unsigned int word, int length, unsigned int mask)

2.2.6 Controllo uniformità ed incremento istogramma - lato device

Nel caso il conteggio sia minore od uguale a 2 il LBP è uniforme, l'uniformità ed eventualmente il numero di bit ad 1 del LBP uniforme definiscono quale colonna dell'istogramma incrementare.
Quindi nel caso positivo si torna l'indice di colonna dell'istogramma, pari al conteggio degli 1 del LBP (sempre tramite __popc(unsigned int x)), altrimenti si torna l'indice della colonna riservata alla frequenza dei LBP non uniformi, pari a *points* + 1. Infine il thread incrementa l'istogramma utilizzando la funzione atomica atomicAdd per evitare race conditions.

LocalBinaryPatternGPUShared.cu:

- __device__ static unsigned int calculateHistogramIndex(unsigned int LBPValue, unsigned int numberOfBitSwitches, int points)

- __device__ static void incrementBin(Histogram histogram, unsigned int binIndex)

2.3 Implementazione tramite texture memory

L'algoritmo LBP accede ai pixel dell'immagine seguendo uno schema spaziale, inoltre l'interpolazione bilineare è una funzionalità che viene fornita automaticamente se si utilizza la texture memory.
Per questi motivi si è voluto implementare un kernel che utilizza la texture memory, in modo da confrontare i risultati ottenuti con la soluzione che usa la shared.
Poichè molte parti del codice sono identiche alla versione con shared memory verranno riportate solo le differenze principali, che riguardano perlopiù l'utilizzo di funzioni e costrutti CUDA diversi.

2.3.1 Differenze d'implementazione - lato host

Bisogna creare un oggetto di tipo texture che fa riferimento ad essa e serve sia per configurare le opzioni relative alla texture nel codice host, sia per fare riferimento alla texture nel codice device.
Durante la sua creazione bisogna specificare:

- Type = unsigned char
  il tipo di dati nel buffer di partenza, in questo caso unsigned char
- Dim = cudaTextureType2D
  la dimensionalità della texture, in questo caso 2D
- ReadMode = cudaReadModeNormalizedFloat
  il formato dei dati letti nel kernel, l'intensità dei pixel è un float compreso tra 0.0f ed 1.0f

Prima del lancio del kernel è necessario specificare altri campi dell'oggetto texture:

- addressMode[0] = addressMode[1] = cudaAddressModeBorder
  la modalità di lettura nel caso si legge fuori dai bordi della texture, in questo caso si usa 0-padding (l'indice specifica se ci si riferisce alla dimensione x o y).
- filterMode = cudaFilterModeLinear
  si vuole interpolare linearmente i valori che non ricadono al centro di un texel, avendo specificato precedentemente la dimensionalità 2D si applica l'interpolazione bilineare

L'utilizzo della texture memory richiede metodi d'allocazione della memoria e di trasferimento dei dati diversi.

```
cudaError_t cudaMallocPitch(void** devPtr, size_t* pitch, size_t width, size_t height
    ↪ )
```

Alloca la memoria e allinea le righe aggiungendo automatico dei byte di padding dove serve.

```
cudaError_t cudaMemcpy2D(void* dst, size_t dpitch, const void* src, size_t spitch,
    ↪  size_t width, size_t height, enum cudaMemcpyKind kind)
```

Trasferisce dati da host a device.

```
cudaError_t cudaBindTexture2D(size_t* offset, const textureReference* texref,
    ↪ const void* devPtr, const cudaChannelFormatDesc desc, size_t width,
    ↪ size_t height, size_t pitch)
```

Infine la referenza alla texture lato host va bindata alla zona di memoria allocata sulla GPU.

2.3.2 Differenze d'implementazione - lato device

Non usando più la shared memory, il codice che la inizializzava è stato rimosso, ora i dati riguardanti l'immagine vengono letti direttamente dalla texture memory tramite la funzione:

```
tex2D(textureReference, x, y)
```

A cui si passa il riferimento alla texture e le coordinate del pixel, traslate di 0.5f, poichè CUDA si aspetta che il centro sia a .5.

2.4 Differenza di risultati tra implementazione shared memory e texture memory

Gli istogrammi generati da shared memory e texture memory usando l'interpolazione bilineare divergono ma sono abbastanza simili.
Si pensa che alcuni fattori legati alla precisione contribuiscano a queste diversità, tra cui il fatto che i valori α e β utilizzati nel calcolo dell'interpolazione tramite texture memory sono rappresentati con 9 bit a virgola fissa di cui 1 per il segno ed 8 per il valore decimale (documentazione CUDA) mentre nel kernel shared vengono rappresentati col tipo float.
Si è giunti a queste conclusioni perchè se si esegue il LBP utilizzando il nearest neighbor invece dell'interpolazione bilineare i tre istogrammi (CPU, kernel shared e kernel texture) sono identici.
Un'altra precisazione da fare è che per ottenere risultati identici tra il kernel shared e la CPU utilizzando l'interpolazione bilineare è necessario disattivare l'ottimizzazione fused-multiply-add (-fmad=false) durante la compilazione.

3 Risultati ottenuti

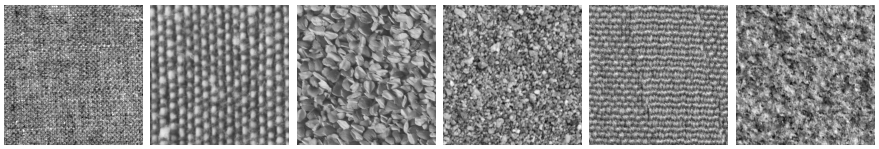
3.1 Introduzione

L'implementazione è stata testata in due modi, nel primo caso per valutarne la capacità di classificazione, nel secondo per confrontare le tempistiche tra diverse versioni.
L'algoritmo LBP è parametrico, quindi in entrambi i test si sono usate più coppie di parametri.
I test sono stati eseguiti su una macchina con le seguenti caratteristiche:

- Intel Core i7-4702MQ 2.2 GHz
- 8 GB RAM DDR3
- Nvidia GT 750M 4 GB
- Windows 10 Pro 64 bit
- Cuda 9.2

3.2 Capacità di classificazione

Per testare la capacità di classificazione dell'algoritmo si è utilizzato un dataset composto da textures grayscale con risoluzione 576 x 576 suddivise in sei classi, ognuna contenente 40 texture con angoli di rotazione diversi, per un totale di 240 immagini. Per ogni classe si è scelta un'immagine come rappresentante e si sono calcolati i loro istogrammi.
Successivamente si è fatto la stessa operazione per tutte le 240 immagini del dataset, confrontando ogni istogramma generato con i sei di riferimento.
L'esperimento è stato ripetuto variando i valori dei parametri *points* e *radius* ed i risultati della classificazione sono riportati nelle matrici di confusione successive.
Osservando i risultati si può notare come si riesce a classificare le texture in modo abbastanza corretto al variare dei parametri.
In particolare abbiamo le classi canvas e seat dove tutti gli elementi sono classificati correttamente indipendentemente dai parametri, anche linsseeds ottiene buoni risultati con tutte le configurazioni.
Alcune classi vengono classificate bene con una certa configurazione mentre con altre in modo meno preciso, ad esempio stone viene classificato bene con (16, 2) e (24, 3), ma con (8, 1) i risultati sono molto meno buoni.
Cushion si comporta abbastanza bene con (8, 1) ma peggiora con le altre coppie di parametri.
Infine abbiamo la classe sands dove con qualsiasi configurazione c'è dell'incertezza con stone.
In conclusione l'algoritmo ha una buona capacità di classificazione, che potrebbe essere probabilmente migliorata combinando i risultati derivanti dalle diverse parametrizzazioni.



(a) Canvas (b) Cushion (c) Linsseeds (d) Sand (e) Seat (f) Stone

Figura 2: le sei classi di texture utilizzate

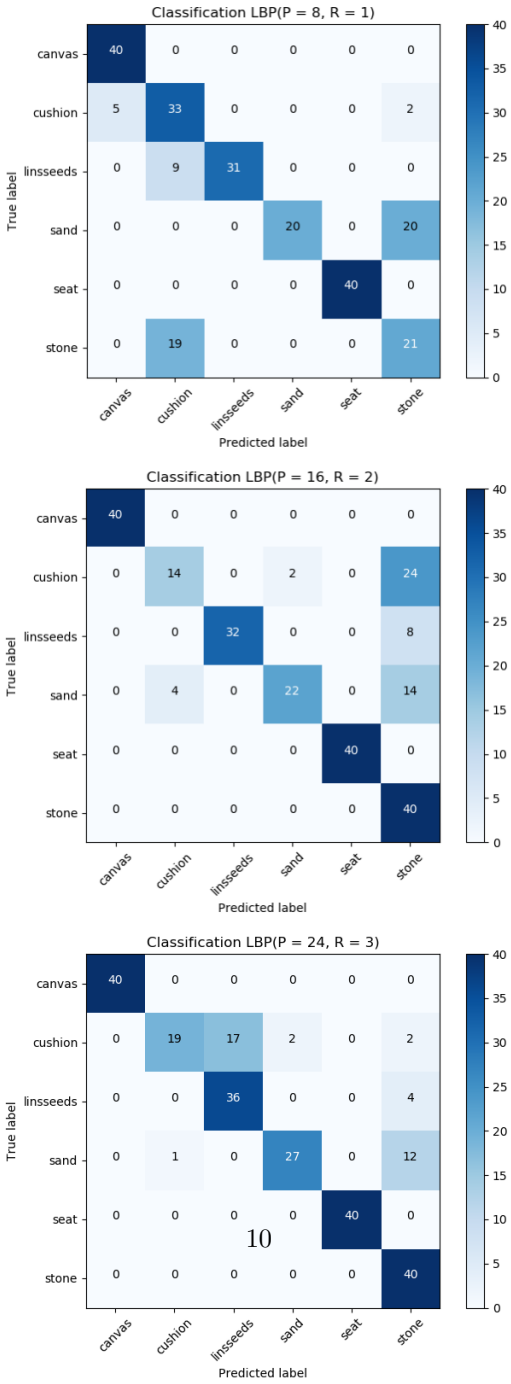


Figura 3: matrici di confusione per tre coppie di parametri che mostrano la classificazione delle texture del dataset, le righe sono l'etichetta giusta e le colonne

3.3 Tempi d'esecuzione e speedup rispetto alla versione sequenziale

L'algoritmo è stato eseguito su una stessa immagine a 5 risoluzioni diverse, ognuna il quadruplo della precedente, per ogni implementazione (cpu, shared memory e texture memory) si è calcolato il tempo medio su 10 esecuzioni.
I tempi dei kernel sono stati registrati tramite eventi CUDA e comprendono l'allocazione della memoria device, il trasferimento iniziale dell'immagine e finale dell'istogramma e l'esecuzione dell'algoritmo.
I tempi d'esecuzione su CPU comprendono l'esecuzione dell'algoritmo sull'immagine allocata nell'heap.
Come si vede dai risultati ottenuti l'algoritmo è altamente parallelizzabile poichè ogni LBP viene calcolato indipendentemente dagli altri, questo si ripercuote sui valori di speedup ottenuti, infatti indipendentemente dal valore dei parametri utilizzati e dalla risoluzione dell'immagine si ha sempre uno speedup significativo, che aumenta sia al crescere dei parametri dell'algoritmo sia alla dimensione della texture.
L'utilizzo della texture memory risulta essere la migliore soluzione già a partire da risoluzioni piuttosto basse probabilmente a causa del pattern spaziale con cui si accede ai pixel, infatti a parte la risoluzione minore dove i risultati sono inferiori a quelli della shared memory, il divario di prestazioni tende ad aumentare velocemente.
Si sono usate texture molto grandi per osservare l'andamento dello speedup, con la texture memory lo speedup sembrerebbe avere ancora margini di crescita all'aumentare della risoluzione, la shared memory invece sembra essere prossima a giungere ad un limite.
Si può concludere dicendo che il calcolo del LBP su GPU offre enormi vantaggi in termini di prestazioni, il fatto di utilizzarlo in combinazione con l'interpolazione bilineare è molto oneroso se eseguito su CPU a differenza della GPU.
Oltre alla classificazione di textures il LBP ha altre applicazioni, ad esempio il riconoscimento facciale, l'implementazione tramite GPU rende possibile esaminare frames ad alte risoluzioni anche in contesti real time.

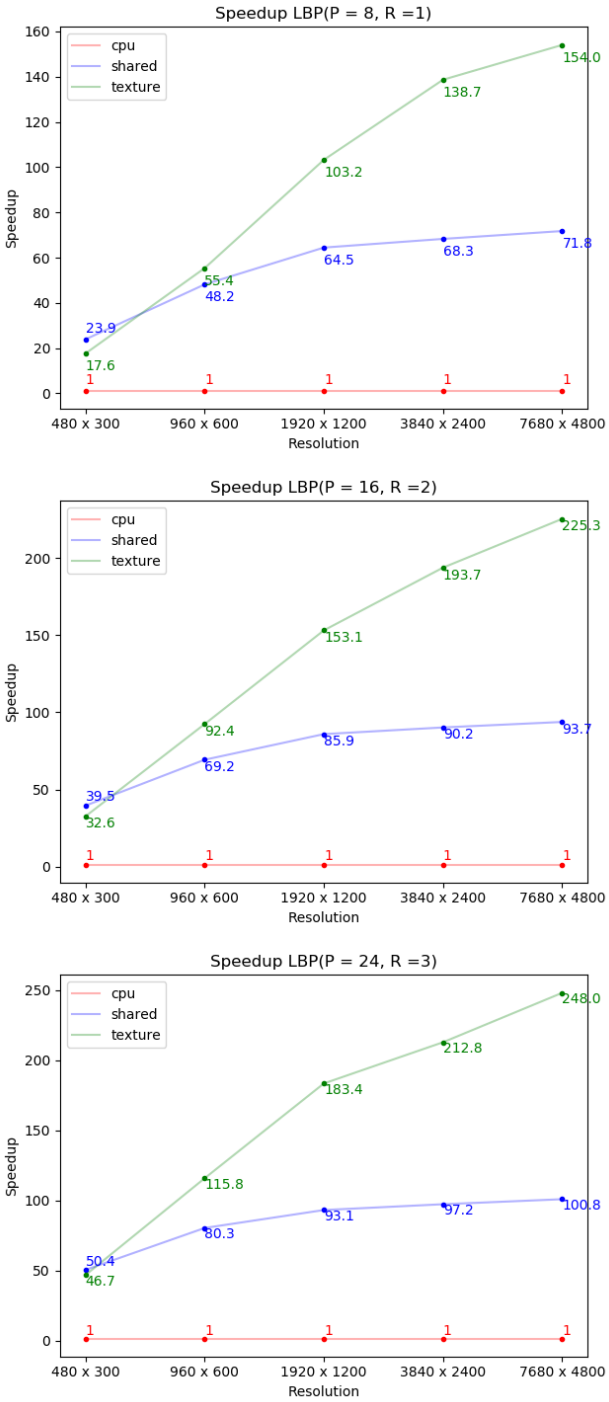


Figura 4: grafici degli speedup ottenuti variando i parametri dell'algoritmo

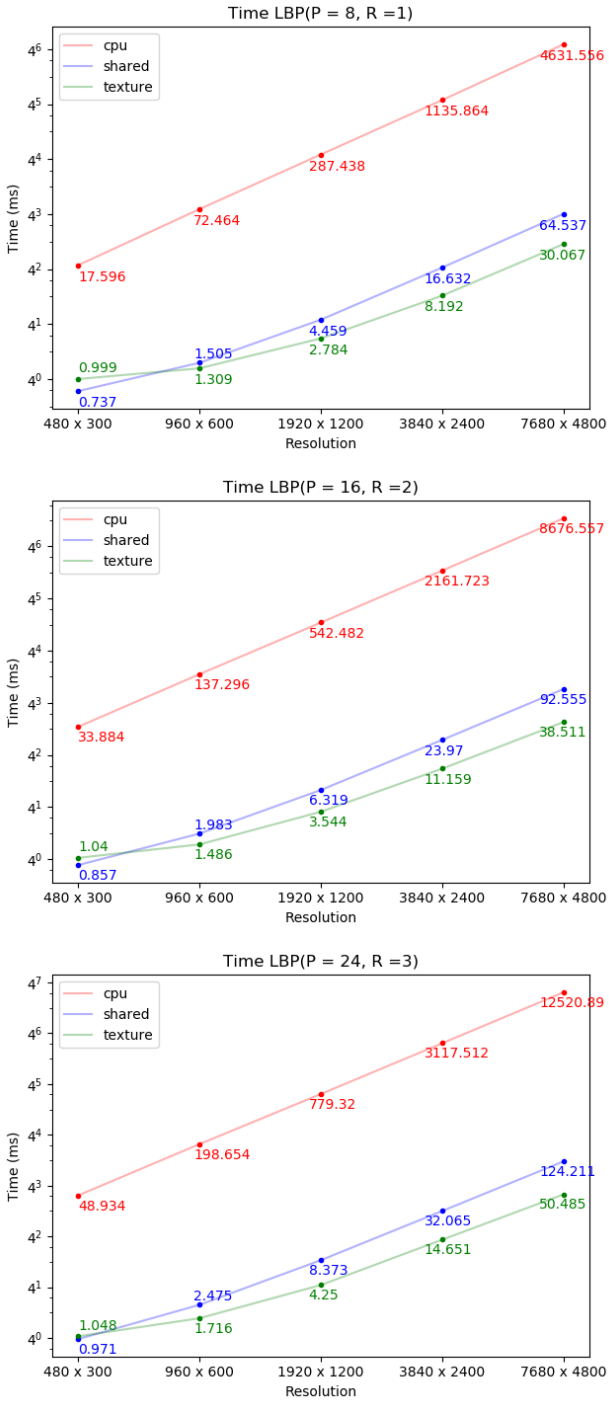


Figura 5: grafici dei tempi ottenuti variando i parametri dell'algoritmo, le scale sono logaritmiche

4 Istruzioni per compilare ed eseguire il programma

Il programma va compilato a 64 bit sotto Windows, per assicurarsi che i risultati ottenuti tra CPU e shared siano uguali bisogna specificare l'opzione -fmad=false durante la compilazione con nvcc.
È richiesta la libreria FreeImage, può essere scaricata dal sito ufficiale.
Il programma può essere eseguito in due modalità:

1. Esecuzione del LBP su una texture tramite:

   nomeProgramma "path" device points radius label

   - path: il path assoluto della texture, nel formato "C:\\path\\file\\nomeFile.ext"
   - device: permette la scelta tra l'esecuzione tramite CPU, kernel shared o kernel texture, specificando rispettivamente cpu, shared o texture
   - points: il parametro *points* del LBP, un numero intero
   - radius: il parametro *radius* del LBP, un numero decimale
   - label: un'etichetta che indica la categoria dell'immagine, se si vuole solamente eseguire l'algoritmo specificare una stringa casuale

   Al termine dell'esecuzione se non esistono vengono creati i file times.csv e model_p{*points*}_r{*radius*}.csv a cui viene aggiunta una riga di testo contenente il tempo d'esecuzione e l'istogramma generato.

2. Esecuzione delle funzioni di test dichiarate nella classe Test.h tramite:

   nomeProgramma "path"

   - path: il path alla cartella contenente le immagini necessarie (vedi spiegazione sottostante), nel formato "C:\\path\\cartella\\"

   Il primo test calcola gli istogrammi su immagini di test predefinite, questi sono confrontati con quelli calcolati nelle prime fasi dello sviluppo, in modo da assicurarsi di non avere introdotto errori nelle fasi successive.
   Se si volesse evitare il primo test si specifica come parametro un path non esistente ("path\\a\\caso"). Il secondo test genera delle immagini casuali a diverse risoluzioni e si confrontano gli istogrammi di cpu e shared per controllare che i risultati siano uguali, per la texture non è possibile (sezione 2.4).
   In ognuno dei test viene controllato che la somma delle frequenze dell'istogramma sia uguale al numero di pixel dell'immagine che lo genera.

La lettura dei file contenenti gli istogrammi e la classificazione tramite la formula di likelihood **??** viene effettuato tramite lo script python classify.py:

python classify.py "C:\\path\\file\\modelli.csv" "C:\\path\\file\\istogrammi.csv"