Marco Ciccone
Architetture Avanzate A/A 2011/2012
Prof. Claudio Lucchese
08/07/2012

Implementazione degli algoritmi K-Means e Mean Shift con nVidia CUDA

Il progetto d'esame consiste nell'implementazione degli algoritmi di clustering Kmeans e Mean Shift per mezzo CUDA (acronimo di Compute Unified Device Architecture). CUDA è un'architettura hardware per l'elaborazione parallela creata da nVidia. Tramite l'ambiente di sviluppo per CUDA, è possibile scrivere applicazioni capaci di eseguire calcolo parallelo sulle GPU delle schede video nVidia.

K-Means

Strutture dati utilizzate

```
typedef struct {
    float x; // coordinata x del centroide
    float y; // coordinata y del centroide
    int index_cluster; // cluster di appartenza
}valpoint;

typedef struct {
    float x; // coordinata x del punto
    float y; // coordinata y del punto
    unsigned long numMembers; // numero di punti all'interno del cluster del centroide
}centroid;
```

Parametri

K (numero dei cluster)
Nome del file di input

Implementazione

1. Caricamento dei punti dal file di testo
2. Si scelgono K punti random e si utilizzano per inizializzare la struttura dati array di centroidi.
3. Si esegue il kernel CUDA dell'algoritmo con un numero di thread pari al numero di punti :
4. Al termine dell'esecuzione del kernel si verifica quanto è lo scostamento fra i centroidi appena calcolati e i centroidi calcolati all'iterazione precedente. Se lo scostamento è molto piccolo allora significa che la procedura può terminare. Altrimenti eseguo nuovamente il Kernel.

Analisi del kernel

Per ogni valore della lista (uno per thread) viene calcolata la distanza con il centroide di ogni cluster.
Calcolando le distanze controllo quindi di quale cluster fa parte il punto e glielo assegno.
Si attende che tutti i thread abbiano finito la procedura di assegnazione dei punti ai cluster
Quando tutti i thread hanno finito la procedura è possibile parallelizzare il ricalcolo del centroide, assegnando il lavoro solo ai thread con id compreso fra 0 e K (numero Kluster).

In questo modo il lavoro dei thread sarà sbilanciato , dato che il numero dei cluster è << del numero dei punti, ma ho comunque parallelizzato un'operazione dispendiosa.

Mean Shift

Strutture Dati utilizzate

```
typedef struct {
    float x; // coordinata x del punto
    float y; // coordinata y del punto
    int index_cluster; // cluster di appartenza
    int start; // indice di partenza dei punti distanti max H dal punto
    int end; // indice di fine dei punti distanti max H dal punto
}valpoint;

typedef struct {
    float x; // coordinata x del centroide
    float y; // coordinata y del centroide
    unsigned long iterations; // numero di iterazioni per arrivare alla convergenza
    float meanshift; //scostamento con il centroide precedentemente calcolato
}centroid;
```

Parametri

H (bandwidth)
Nome del file di input

Implementazione

1. Caricamento dei punti dal file di testo
2. Scrematura dei punti ripetuti in modo da diminuire la complessità dell'algoritmo
3. Si esegue il kernel CUDA dell'algoritmo con un numero di thread pari al numero di punti
4. Al termine della procedura che ha calcolato il centroide per ogni punto, si raggruppano i punti in cluster in base alla vicinanza dei centroidi.

Analisi del kernel:

Per ogni valore della lista (uno per thread) viene calcolato il centroide (gradiente)
Il centroide viene calcolato solo con i punti che stanno entro un raggio H dal punto.
Si ripete la procedura di calcolo fino a che il meanshift non converge a 0.

$$y_i^{t+1} = \frac{\sum_{i=1}^{n} x_j e^{\frac{-|y_i^t - x_j|^2}{h^2}}}{\sum_{i=1}^{n} e^{\frac{-|y_i^t - x_j|^2}{h^2}}}$$

Nel nostro caso abbiamo utilizzato un Kernel Gaussiano.

Difficoltà riscontrate durante il progetto

A causa dei limiti prestazionali della GPU e della complessità computazionale dell'algoritmo, ho incontrato non poche difficoltà durante l'implementazione del mean shift.

1. Non è stato possibile calcolare il mean shift su tutti i punti nello stesso kernel. Per questo motivo la procedura è stata suddivisa in più parti,
ogni kernel calcola il centroide per 1024 punti.
In questo modo si evita che la GPU restituisca un errore di timeout a causa del troppo tempo richiesto per l'esecuzione.
Ho evitato di modificare il tempo di timeout perchè il controllo è necessario nel caso in cui ci siano loop.
Su windows pare che basti bloccare lo watchdog. Su Ubuntu c'è un'altra procedura da fare sull' X server, ma non ho approfondito.
Il tempo di timeout comunque aumenta parecchio se si utilizza una GPU dedicata. Nel caso in cui si utilizzi una GPU che gestisce anche il display
è inferiore per ovvi motivi.

http://stackoverflow.com/questions/497685/how-do-you-get-around-the-maximum-cuda-run-time
http://stackoverflow.com/questions/5331140/setting-cudadeviceproperty-cudakernelexectimeoutenabled

2. La procedura di calcolo del centroide per ogni punto non viene fatta utilizzando tutti i punti, ma solo quelli che stanno entro un raggio H.
In questo modo diminuisce molto il tempo di calcolo .
Per trovare i punti che stanno all'interno del raggio ho ordinato l'array dei punti in base alla coordinata y e per ogni punto ho calcolato l'indice dei due punti
che delimitavano una finestra di altezza max H.
All'interno del kernel ho scremato ancora i punti prendendo solo quelli che rientravano in un cerchio di raggio H dal punto di cui stavo calcolando il centroide.

3. Ho riscontrato il seguente errore: "too many resources requested for launch", dopo una ricerca ho scoperto che l'errore è dovuto al fatto che si stanno superando i
registri disponibili perchè la dimensione dei blocchi è troppo grande quindi ho abbassato da 512 a 256.

http://stackoverflow.com/questions/7584965/out-of-resources-error-while-doing-loop-unrolling

4. Quando vengono raggruppati i punti in base al centroide che è stato calcolato c'è un'approssimazione.
Come si può vedere dal plot dei punti , i centroidi non sono sempre precisi.

5. Non sono riuscito a fare dei test modificando la funzione di Kernel gaussiana a causa dei soliti problemi di risorse.
Nei papers trovati in rete infatti c'è un H^2 a denominatore, ma sembra funzionare bene lo stesso.

Per incrementare drasticamente le performance dell'algoritmo di mean shift sono stati effettuati studi riguardanti l'utilizzo di strutture dati ad albero come il kd-tree.
Altro tipo di struttura dati che è stata utilizzata è una tabella Hash, per recuperare velocemente i punti vicini al punto di cui calcolare il centroide (neighbours).

Questi tipo di strutture dati sono consigliati quando si ha a che fare con milioni di punti a dimensione molto elevate, per questo motivo e per la complessità
di realizzazione non sono state prese in considerazione per il progetto.

Per quanto riguarda il kmeans non ho riscontrato particolari problemi.

Per analizzare meglio i risultati prodotti ho pensato fosse meglio avere un riscontro grafico.
Per questo motivo ho utilizzato le librerie OpenGL per disegnare i punti, e colorarli in base al cluster di appartenenza.

Le funzioni utilizzate sono le seguenti:

GLvoid InitGL(GLvoid);
Inizializza la finestra OpenGL, colore, dimensione, profondità etc...

GLvoid DrawGLScene(GLvoid);
E' la funzione che si occupa di disegnare i punti e i centroidi.
Non è possibile passare parametri, quindi le variabili utilizzate all'interno devono essere dichiarate static.

```
static valpoint* h_idata;
static centroid* h_centroids;
static unsigned long numElements;
static unsigned long numClusters;
```

GLvoid glCircle3f(GLfloat x, GLfloat y, GLfloat radius);
Disegna un cerchio. Serve per identificare il centroide

Risultati e tempi di esecuzione

Tutti i test sono stati eseguiti su un Notebook HP Pavilion 6690el con GPU nVidia GeForce 8400M GS e sistema operativo Ubuntu 10.10.
La capacità di calcolo della GPU è la versione 1.1 secondo gli standard CUDA.

Mean Shift

Per quanto riguarda il mean shift sono stati effettuati test con il parametro H per il file firenze.gps
I valori che H può assumere vanno da 0.001 a 0.005, negli altri casi non si ottiene dei buoni risultati.

Con H = 0.001 si ottengono 87 cluster in 7.088 secondi
Con H = 0.002 si ottengono 81 cluster in 14.558 secondi
Con H = 0.003 si ottengono 78 cluster in 18.83 secondi
Con H = 0.004 si ottengono 72 cluster in 22.89 secondi
Con H = 0.005 si ottengono 66 cluster in 26.667 secondi

Da 0.006 in poi i valori cominciano a essere sballati.
Viene calcolato un grosso cluster centrale che non sembra corretto.

K-means

Per quanto riguarda il kmeans i test sono stati effettuati variando il numero dei cluster K per il file Firenze.gps

La soglia di convergenza è fissata a 0.01.

Con K = 10 si ottengono 16 iterazioni prima della convergenza, eseguite in 1.610 secondi
Con K = 10 si ottengono 9 iterazioni prima della convergenza, eseguite in 0.957 secondi
Con K = 10 si ottengono 3 iterazioni prima della convergenza, eseguite in 0.398 secondi

Con K = 20 si ottengono 13 iterazioni prima della convergenza, eseguite in 1.569 secondi
Con K = 20 si ottengono 11 iterazioni prima della convergenza, eseguite in 1.3005 secondi
Con K = 20 si ottengono 7 iterazioni prima della convergenza, eseguite in 0.869 secondi

Con K = 30 si ottengono 30 iterazioni prima della convergenza, eseguite in 3.910 secondi
Con K = 30 si ottengono 19 iterazioni prima della convergenza, eseguite in 2.524 secondi
Con K = 30 si ottengono 18 iterazioni prima della convergenza, eseguite in 2.397 secondi

Con K = 30 si ottengono 30 iterazioni prima della convergenza, eseguite in 3.910 secondi
Con K = 30 si ottengono 19 iterazioni prima della convergenza, eseguite in 2.524 secondi
Con K = 30 si ottengono 18 iterazioni prima della convergenza, eseguite in 2.397 secondi

Con K = 40 si ottengono 30 iterazioni prima della convergenza, eseguite in 4.099 secondi
Con K = 40 si ottengono 25 iterazioni prima della convergenza, eseguite in 3.47 secondi
Con K = 40 si ottengono 9 iterazioni prima della convergenza, eseguite in 1.313 secondi

Con K = 50 si ottengono 15 iterazioni prima della convergenza, eseguite in 2.179 secondi
Con K = 50 si ottengono 10 iterazioni prima della convergenza, eseguite in 1.583 secondi
Con K = 50 si ottengono 9 iterazioni prima della convergenza, eseguite in 1.38 secondi

Con K = 60 si ottengono 36 iterazioni prima della convergenza, eseguite in 5.504 secondi
Con K = 60 si ottengono 19 iterazioni prima della convergenza, eseguite in 2.951 secondi
Con K = 60 si ottengono 32 iterazioni prima della convergenza, eseguite in 4.86 secondi

Con K = 70 si ottengono 11 iterazioni prima della convergenza, eseguite in 1.8607 secondi
Con K = 70 si ottengono 21 iterazioni prima della convergenza, eseguite in 3.476 secondi
Con K = 70 si ottengono 12 iterazioni prima della convergenza, eseguite in 2.039 secondi

Con K = 80 si ottengono 8 iterazioni prima della convergenza, eseguite in 1.476 secondi
Con K = 80 si ottengono 11 iterazioni prima della convergenza, eseguite in 1.992 secondi
Con K = 80 si ottengono 13 iterazioni prima della convergenza, eseguite in 2.394 secondi

Con K = 90 si ottengono 9 iterazioni prima della convergenza, eseguite in 1.738 secondi
Con K = 90 si ottengono 12 iterazioni prima della convergenza, eseguite in 2.279 secondi
Con K = 90 si ottengono10 iterazioni prima della convergenza, eseguite in 1.909 secondi

La soglia di convergenza è stata fissata a 0.01 per evitare che l'esecuzione del programma durasse troppo.
Si ottengono comunque dei buoni risultati.
I tempi di esecuzione sono variabili, dovuto anche al fatto che i centroidi inizialmente vengono scelti random.

Con il file di esempio Lucca.gps i tempi di esecuzione sono molto più bassi, ho evitato di inserire i risultati. Verranno mostrati in fase di presentazione

Conclusioni:
Sicuramente i tempi di esecuzione sono migliorabili. Sia come implementazione che grazie a una GPU più potente
Sarebbe stato interessante effettuare un confronto con schede video di diversa potenza, ma non ne avevo a disposizione.
Il kmeans è un algoritmo molto più veloce che necessita come parametro di partenza il numero di cluster in cui suddividere lo spazio dei punti.
Il Mean Shift è un algoritmo abbastanza lento, soprattutto se aumenta la dimensione dei punti.
Inoltre va scelto bene il parametro H, che determina la Banda entro il quale calcolare il centroide.
Se si sceglie troppo grande o troppo piccolo l'accuratezza dei risultati e il tempo di esecuzione rischiano di essere non accettabili.
Per questo motivo sono state effettuate diverse prove per ottenere risultati soddisfacenti.