Linear discriminant analisys con CUDA C

Francesco Polvere *

4 dicembre 2018

Sommario

In questo documento viene presentata una implementazione dell'algoritmo Linear Discriminant Analisys tramite l'utilizzo della GPU sulla piattaforma CUDA. L'implementazione parallela viene confrontata con una implementazione classica in C, tramite la misura di tempo di esecuzione ed il calcolo dello speed-up.

1 Analisi del modello teorico

LDA è un metodo utilizzato per trovare una combinazione lineare di feature che caratterizzato o separano due o più classi di oggetti o eventi. La combinazione risultante può essere utilizzata come classificatore lineare o ancor prima della classificazione, come operazione di riduzione di dimensionalità.

1.1 LDA Multiclasse

Supponiamo di avere n classi. La Within-scatter matrix è calcolata come nella formula 1

$$Sw = \sum_{i=1}^{C} \sum_{x \in Ci} (x - x_i)(x - x_i)' \tag{1}$$

Dove x è il vettore osservazione preso dall'insieme Ci mentre xi è il vettore media della i-esima classe.

La between-scatter matrix è calcolata seguendo la formula 2

$$Sb = \sum_{i=1}^{n} m_i(\overline{x}_i - \overline{x})(\overline{x}_i - \overline{x})' \tag{2}$$

Dove $\overline{x}$ é il vettore media totale di tutte le classi e m_i è l'i-esimo vettore della classe C_i.

*F. Polvere, Corso di GPU computing, A/A 2017-2018, Università degli studi di Milano, via Celoria 28, Milano, Italia
E-mail: francesco.polvere@studenti.unimi.it

Dopo aver ottenuto Sb e Sw, vogliamo trovare l'equazione lineare che massimizza l'equazione in figura.

$$J(W) = \frac{|W^T S_b W|}{|W^T S_w W|} \tag{3}$$

Si può dimostrare che la trasformazione W può essere ottenuta risolvendo un problema sugli autovalori.

$$S_b W = \lambda S_w W \tag{4}$$

2 Progettazione ed implementazione

2.1 Introduzione

Da una analisi del metodo di LDA, si può notare la presenza di numerose operazioni su matrici che presentano un alto grado di parallelizzabilità.
Durante la progettazione dell'algoritmo, ho diviso lo sviluppo in sei fasi eseguite in sequenza.

- Calcolo media delle classi e media totale
- Calcolo della Between-scatter matrix
- Calcolo della Within-scatter matrix
- Calcolo della matrice inversa
- Calcolo degli autovettori e autovalori
- Trasformazione dei dati di partenza

2.2 Assunzioni e notazioni utilizzate

Si assume che i dati siano stati estratti da una distribuzione gaussiana ed il numero di osservazioni sia uguale per ogni classe.
Nel documento utilizzerò le seguenti notazioni.

- C: numero di classi (o matrici)
- N: numero di osservazioni (o righe della matrice)
- M: numero di feature (o colonne della matrice)

2.3 Parallelizzazione di LDA utilizzando CUDA

Il dataset utilizzato è diviso in tre file, uno per ogni classe, contenente le osservazioni per ognuna di esse. Una volta caricati in memoria host, vengono copiati sulla sulla memoria device, in particolare nella global memory, il cui spazio è rilasciato solo al termine dell'esecuzione dell'intero programma.
L'allocazione su host è di tipo *pinned* per permettere l'esecuzione sovrapposta del caricamento da host a device di ogni classe ed il calcolo della media della classe stessa.

2.4 Calcolo media relativa e totale

Media della i-esima classe Per ogni classe viene eseguita una copia dei dati dalla memoria host a quella device tramite l'utilizzo della funzione *cudaMemcpyAsync*. Successivamente tramite il kernel ***matrix_mean*** viene calcolata la media della matrice. Questi C-vettori avranno dimensione 1xM dove M è il numero di feature
L'utilizzo di stream pari al numero di matrici da elaborare permette overlapping tra l'operazioni I/O e di computazione, e anche ovelapping tra computazioni nel caso di piccolo kernel come mostrato in figura 1.
I risultati sono mantenuti in global memory e non riportati in memoria host per ottimizzarne le prestazioni.

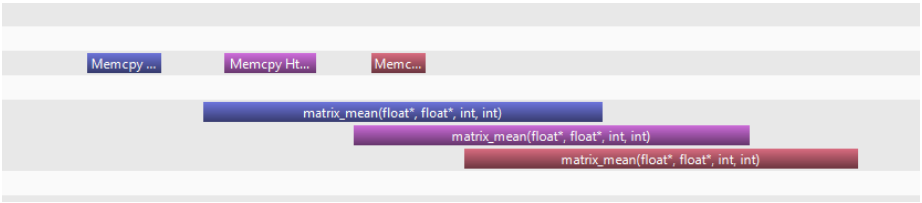

Figura 1: Gli stream per il calcolo della media

Media totale La media totale è calcolata con il kernel ***add_vectors*** che esegue la somma degli C-vettori risultati dal calcolo precedente e utilizzando il kernel ***div_by_scalar*** che esegue la divisione del vettore per uno scalare che è pari al numero di classi.
Naturalmente il vettore media totale avrà dimensione 1xM.

2.5 Between-scatter matrix

Il calcolo della Between-scatter matrix necessita di tre kernel.
Il kernel ***diff_vect*** si occupa di calcolare la differenza tra due vettori di dimensione M, nel caso specifico tra il vettore media locale i-esimo e media globale.

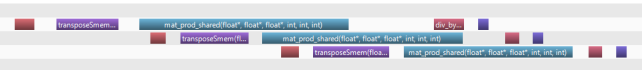

Figura 2: overlapping computazionale

Il kernel viene lanciato su un numero di thread pari alla dimensione dei vettori suddivisi su più blocchi.
Successivamente viene eseguito il kernel ***vector_prod*** per il calcolo del prodotto tra il vettore in uscita dal kernel precedente e la sua trasposta. Il risultato è una matrice quadrata di dimensione M X M.
L'ultimo punto è la somma di tutte le C matrici in uscita dal kernel precedente ed è eseguito dal kernel ***add_matrix***.
Per ogni classe viene eseguito il gruppo dei tre kernel sopra descritti in stream diversi per permettere un livello di esecuzione concorrente maggiore in caso di dimensione del dataset ridotto.

2.6 Within-scatter matrix

Il calcolo della Within-scatter matrix si avvale di cinque kernel qui sotto descritti.

- ***diff_matr_vect***: è il kernel che si occupa di sottrarre il vettore media della classe i-esima dalla i-esima matrice, riga per riga.
- ***transposeSmem***: esegue la trasposizione della matrice in ingresso
- ***mat_prod_shared*** si occupa di eseguire il prodotto tra matrice i-esima trasposta e la stessa matrice non trasposta.
- ***div_by_scalar***: esegue la divisione della matrice in ingresso per uno scalare.
- ***add_matrix***: esegue la delle matrici risultati dalle operazioni precedenti delle C-classi

Essendo piccoli kernel, l'utilizzo di stream pari al numero di matrici mi ha permesso di ottenere un overlapping tra i kernel appartenenti a stream diversi come mostrato in figura 2

2.7 Calcolo matrice inversa di SW

Per il calcolo della matrice inversa ho utilizzato le funzioni *cublasSgetrfBatched()* e *cublasSgetriBatched()* della libreria cuBLAS che eseguono la fattorizzazione LU. Successivamente al calcolo ho eseguito il prodotto tra la matrice inversa ottenuta e la matrice SB, tramite il kernel ***matrix_prod***

2.8 Calcolo autovalori e autovettori

Per il calcolo degli autovalori e degli autovettori ho utilizzato la funzione *cusolverDnDsyevd* della libreria cuSOLVER che esegue la Singular Value Decomposition per il calcolo degli autovettori e degli autovalori della matrice quadrata in ingresso Mat^{MxM}

2.9 Calcolo dei nuovi dati

Una volta ottenuto autovettori e rispettivi autovalori al passo precedente, ho selezionato gli C-1 autovettori che hanno gli autovalori massimi.
Ho quindi moltiplicato la matrice originale, e.g. nel caso di 4096 campioni $Mat^{4094x128}$ per questi C-1 autovettori tramite il kernel ***mat_prod_shared*** ottenendo così la nuova proiezione. La nuova matrice avrà quindi dimensione $Mat^{4096x(C-1)}$
Ho utilizzato un numero di stream pari al numeri di matrici e *cudaMemcpyAsync* per sovrapporre la computazione del kernel che calcola il prodotto matriciale e la copia dei dati dalla device memory alla host memory per poter successivamente eseguire il plot a video.

2.10 Plotting

Per la presentazione del grafico ho utilizzato **Gnuplot** in tutti e due gli algoritmi.
In figura 3 è mostrato un grafico dei dati rispetto alle prime due componenti. In alto sono proiettati i dati originali mentre nell'area in basso i dati trasformati.

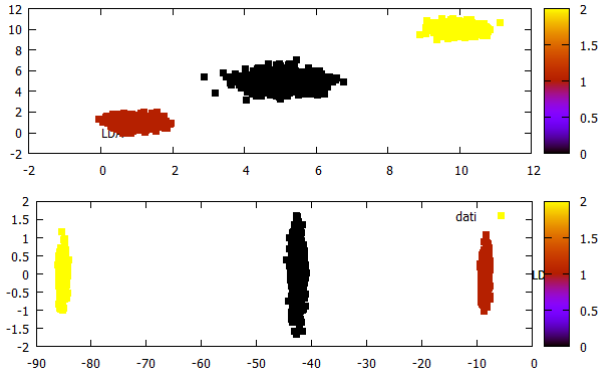

Figura 3: originale e nuova proiezione

3 Simulazione ed esperimenti

Oltre all'algoritmo parallelo presentato qui nella sezione precedente, ho sviluppato anche un algoritmo per LDA sequenziale per verificare l'effettivo speed-up

raggiunto dal primo rispetto al secondo.

3.1 Dataset

Per il benchmark ho utilizzato un dataset generato con una distribuzione normale multivariata. Ogni classe è costituita da 128 feature per campione ed un totale di 4096 istanze per classe.
In termini matematici, ogni classe è rappresentata da una matrice $Mat^{4096X128}$.

3.2 Risultati ottenuti

Il software è stato eseguito su un computer con le seguenti caratteristiche:

- GPU: NVIDIA GeForce 610M, compute capability 2.1, architettura FERMI (2.1) con 48 Cores/SM.
- CPU: Intel(R) Core(TM) i7-3630QM CPU @ 2.40GHz, 2401 Mhz, 4 core, 8 processori logici

Ho eseguito il calcolo del tempo impiegato dall'algoritmo tenendo fissato il numero di classi e di feature e variando il numero di campioni incrementandolo come potenza di due fino al raggiungere il valore massimo del dataset.
Per ogni step, ho eseguito l'algoritmo dieci volte ed elaborato la media dei tempi misurati per cercare di minimizzare l'effetto di eventuali outlier.

In figura 4 è mostrato il grafico dello speed-up della versione parallela rispetto all'implementazine classica. Come è possibile notare, minore è il carico, cioè il numero di osservazioni che l'algoritmo deve elaborare e minore è la differenza prestazionale tra le due versioni. All'aumentare delle osservazioni, a parità di numero di feature, lo speedup aumenta.

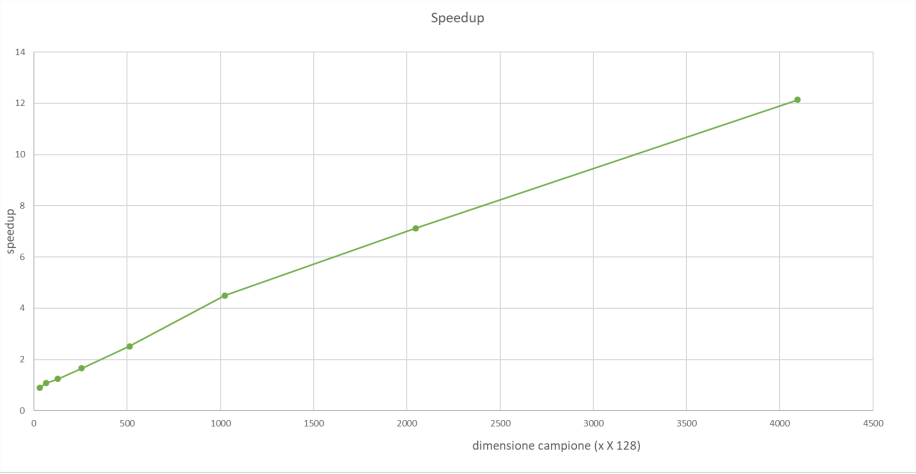

Figura 4: Speed-up

La figura 5 mostra i tempi impiegati dai due algoritmi dove la versione parallela è rappresentata da una quasi-retta orizzontale, mentre la versione sequenziale da una retta inclinata che mostra come aumentano i tempi di calcolo all'aumentare del numero di osservazioni.

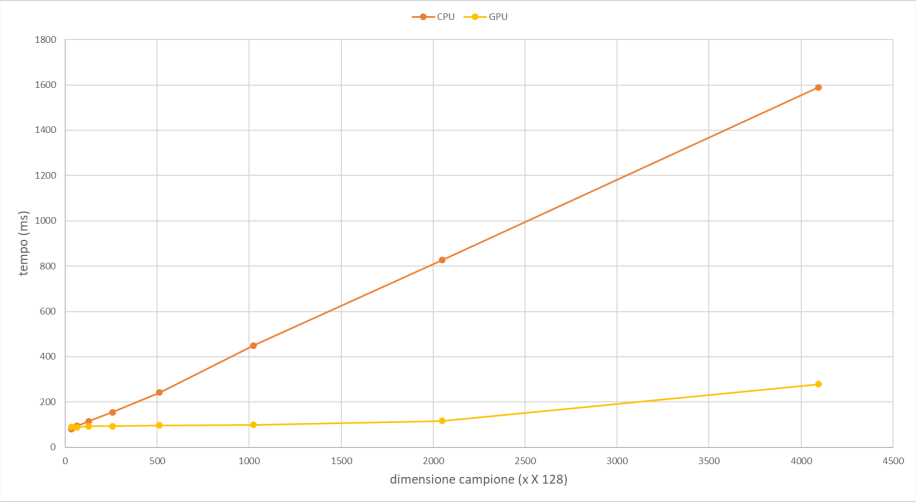

Figura 5: tempi