

# Linear discriminant analysis con CUDA C

Francesco Polvere \*

1 dicembre 2018

## Sommario

In questo documento viene presentata una implementazione dell'algoritmo Linear Discriminant Analysis 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 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.

## 2 Analisi del modello teorico

LDA è un metodo utilizzato per trovare una combinazione lineare di feature che caratterizzino 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.

### 2.1 LDA Multiclasse

Supponiamo di avere  $n$  classi. La Within-scatter matrix è calcolata come:

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

Mentre la between-scatter matrix è calcolata come:

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

---

\*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

Dove  $\bar{x}$  é la media totale di tutte le classi,  $m_i$  é il numero di dati di training per ogni classe.

Dopo aver ottenuto  $S_b$  e  $S_w$ , vogliamo trovare l'equazione lineare che massimizza l'equazione in figura.

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

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

$$S_b W = \lambda S_w W$$

## 2.2 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 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.

**Assunzioni e notazioni** Si assume che i dati siano stati estratti da una distribuzione gaussiana.

Il numero di osservazioni è uguale per ogni classe.

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

**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 `[nome_kernel]` calcola la media della matrice.

L'utilizzo di stream pari al numero di matrici da elaborare permette overlapping tra l'operazioni I/O bound e di computazione, e anche overlapping 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.

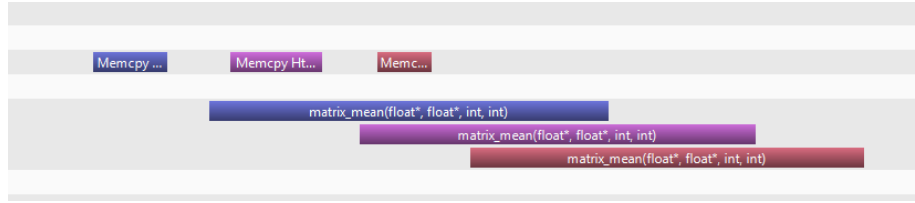


Figura 1: Gli stream per il calcolo della media

**Media totale** La media totale è calcolata con il kernel `jnome_kerneli` che esegue la somma degli C-vettori risultati dal calcolo precedente del kernel `jnome_kerneli` e dividendo per uno scalare pari al numero di classi. Questi n-vettori avranno dimensione  $1 \times 128$  che è pari al numero di feature.

**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. Successivamente si utilizza 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  $M \times M$ .

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.

**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

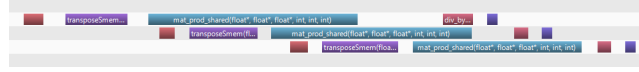


Figura 2: overlapping computazionale

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

**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*

**Calcolo autovalori e autovettori** Per il calcolo degli autovalori e degli autovettori 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  $Mat^{M \times M}$

**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  $Mat^{4094 \times 128}$  per questi C-1 autovettori tramite il kernel (*mat\_prod\_shared* ottenendo così la nuova proiezione. La nuova matrice avrà quindi dimensione  $Mat^{4096 \times (C-1)}$

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 eseguire il plot a video.

**Plotting** Plotto le prime due componenti

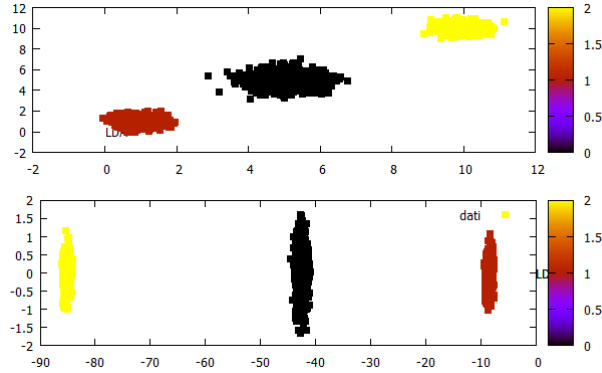


Figura 3: originale e nuova proiezione

### 3 Simulazione ed esperimenti

Ho sviluppato due algoritmi. Quello sequenziale e quello parallelo.

#### 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  $Mat^{4096 \times 128}$ .

#### 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 ed 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'implementazione classica. Come è possibile notare, minore è il carico, il numero di campione che l'algoritmo deve elaborare e minore è la differenza prestazionale tra le due versioni. All'aumentare dei campioni, a parità di numero di feature, aumenta lo speedup.

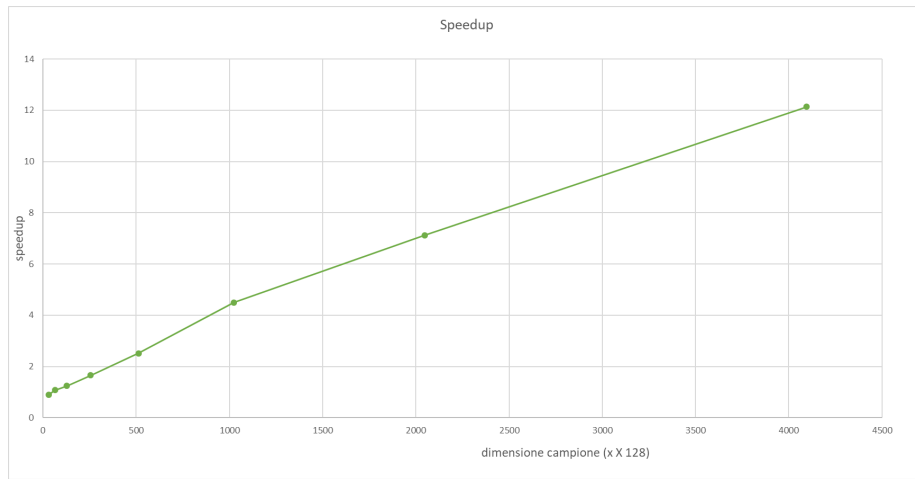


Figura 4: Speed-up

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 campioni.

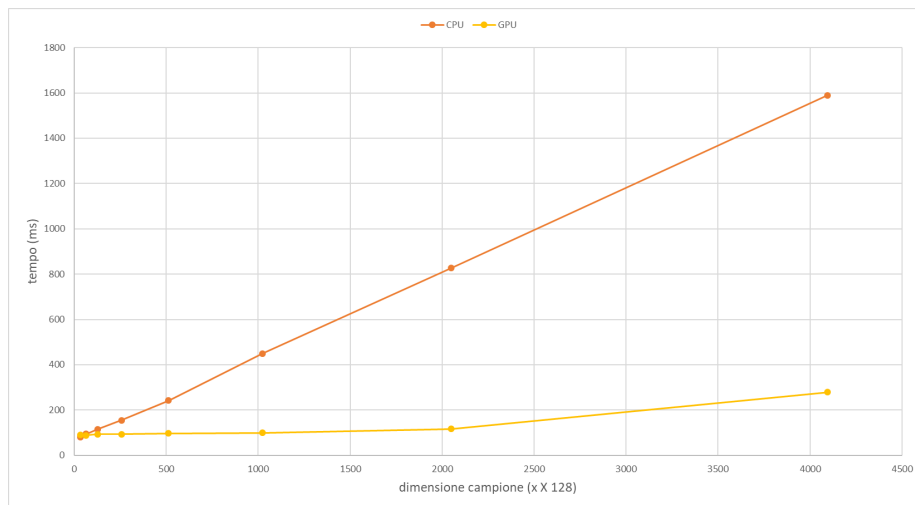


Figura 5: tempi