Università degli Studi di Salerno

Dipartimento di Informatica

Corso di Laurea Magistrale in Informatica

Metodi Numerici per l'Informatica

Seminario

Autore

Simone D'Assisi

Anno Accademico 2025/2026

Indice

1 SGD Parallelo per la Fattorizzazione della Matrice dei Rating nei Sistemi di Raccomandazione

Buongiorno a tutti. In questo seminario presenterò una panoramica dei principali metodi *gradient-based* per la fattorizzazione delle matrici di rating nei sistemi di raccomandazione, concentrandomi però non tanto sugli aspetti modellistici, quanto sui problemi e sulle soluzioni legati alla loro **parallelizzazione su CPU e GPU**.
L'obiettivo è mostrare perché lo **Stochastic Gradient Descent** rappresenti un algoritmo estremamente efficace per la raccomandazione, ma allo stesso tempo difficile da parallelizzare in modo efficiente.

1.1 Introduzione

I sistemi di raccomandazione moderni si basano spesso sulla *matrix factorization*, cioè sulla decomposizione del rating matrix R in due matrici, una matrice U per gli utenti e una V per gli item. L'algoritmo più utilizzato per trovare la migliore decomposizione di R è lo *Stochastic Gradient Descent*, che ha complessità proporzionale al numero di rating osservati. Su dataset reali, parliamo di decine o centinaia di milioni di coppie. Questo implica che un'implementazione sequenziale non è praticabile: la parallelizzazione è obbligatoria. Tuttavia, come vedremo, la struttura stessa della fattorizzazione introduce conflitti tra thread e difficoltà nell'uso di GPU, il che rende il problema molto interessante dal punto di vista della programmazione parallela.

1.2 Richiamo del modello e dell'SGD

Do ora un brevissimo richiamo del modello [1]. La predizione del rating di un utente u su un prodotto v è data da:

$$\hat{r}_{ki} = u_k \cdot v_i,$$

dove u_k è il vettore dei fattori utente e v_i quello dei fattori prodotto. L'errore sul rating osservato è:

$$e_{ki} = r_{ki} - u_k \cdot v_i.$$

Lo **Stochastic Gradient Descent** aggiorna iterativamente i parametri con:

$$u_k = u_k + 2\eta e_{ki} v_i, \qquad v_i = v_i + 2\eta e_{ki} u_k.$$

Con η *learning rate*, che regola l'andamento dell'apprendimento. Questa formula è semplice, ma contiene il primo grande problema dato dalla parallelizzazione: due thread diversi che aggiornano lo stesso utente o lo stesso item generano conflitti. Ed è proprio attorno a questi conflitti che si costruiscono tutte le versioni parallele dell'algoritmo.

1.3 Perché è difficile parallelizzare lo SGD?

Il problema nasce dal fatto che ogni aggiornamento accede in lettura e scrittura a due elementi: u_k e v_i. Se due thread trattano una coppia (u, v) che condivide lo stesso u o lo stesso v, allora accedono allo stesso vettore (questa condizione è definita *race condition*).

Nei dataset reali gli item popolari compaiono migliaia di volte, causando potenziali conflitti continui [2].

1.4 Tecniche di parallelizzazione dello SGD

Passiamo ora alle principali strategie sviluppate per parallelizzare lo SGD.

1.4.1 DSGD (Distributed SGD)

La prima grande idea è il cosiddetto *Distributed SGD*, o DSGD [3]. Si suddivide la matrice dei rating in blocchi, formando una griglia. Si osserva che è possibile selezionare, in ogni iterazione, una collezione di blocchi che non condividono né utenti né item: questa collezione si chiama *strata*. Ogni strata può essere processata in parallelo senza conflitti. Dopo ogni iterazione si passa alla strata successiva, garantendo che tutti i blocchi vengano aggiornati. Questo schema è usato nei sistemi distribuiti perché riduce al minimo la necessità di sincronizzazioni, risultando particolarmente efficace nei sistemi distribuiti su larga scala.

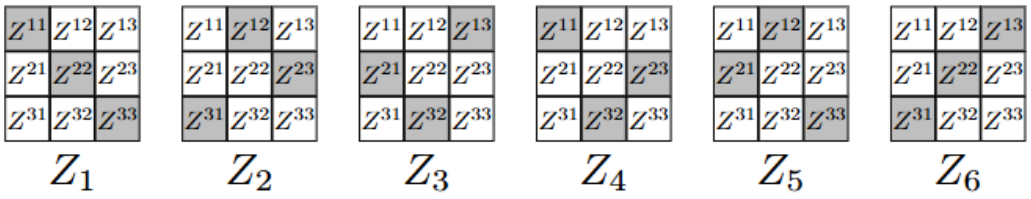

Figura 1: Strata per una divisione in blocchi 3×3 di R.

1.4.2 Hogwild! e metodi lock-free

Il secondo metodo è uno dei più noti per la parallelizzazione di SGD: *Hogwild!* [4]. L'idea è radicale ma estremamente efficace in presenza di dataset molto sparsi. Se la matrice dei rating contiene pochi valori rispetto alle dimensioni totali, allora la probabilità che due thread accedano simultaneamente allo stesso parametro (una riga di U o una riga di V) è estremamente bassa. In questo scenario è possibile eliminare completamente i meccanismi di sincronizzazione, permettendo ai thread di aggiornare i parametri in maniera asincrona e non protetta. L'assenza di lock riduce drasticamente l'overhead di sincronizzazione, consentendo un notevole aumento delle prestazioni rispetto a DSGD e ad altri metodi più strutturati. Tuttavia, Hogwild! non è adatto a tutti i contesti: se il dataset non è abbastanza sparso, oppure se alcuni item o utenti sono molto popolari, il rischio di conflitti e overwriting aumenta, degradando la qualità della soluzione o rallentando la convergenza. Nonostante questi limiti, Hogwild! rimane una delle tecniche più utilizzate per implementazioni parallele su CPU multicore, grazie alla sua semplicità e alla sorprendente robustezza empirica in molti scenari reali.

1.4.3 ASGD e metodi asincroni distribuiti

Un terzo approccio alla parallelizzazione di SGD è l'*Asynchronous SGD* (ASGD), molto usato nei sistemi distribuiti su larga scala. In questo modello, il training è suddiviso tra diversi nodi (o worker), ciascuno dei quali mantiene una copia locale dei parametri e calcola

autonomamente gli aggiornamenti su una porzione dei dati. Gli aggiornamenti sono poi inviati, in modo asincrono, a un nodo centrale (*parameter server*), che li aggrega e aggiorna il modello globale. I worker non attendono che il server confermi l'avvenuto aggiornamento: continuano a eseguire SGD usando parametri che possono essere leggermente "vecchi", fenomeno noto come *staleness*. Questo schema elimina quasi del tutto le sincronizzazioni e consente una scalabilità molto alta, fino a decine o centinaia di macchine. Nonostante la presenza di ritardi negli aggiornamenti, diversi lavori mostrano che ASGD mantiene una buona stabilità e converge efficacemente in molti problemi di ottimizzazione su larga scala [5].

1.5 Parallelizzazione su GPU

Arriviamo ora alla parte più legata alla programmazione parallela su GPU.

1.5.1 Vantaggi teorici

In teoria la GPU è perfetta per questo problema: migliaia di thread che eseguono la stessa formula su dati indipendenti. Tuttavia, nella pratica, l'accesso irregolare alla memoria e il gran numero di conflitti generano difficoltà molto più profonde rispetto a ciò che accade su CPU [6]. I problemi principali sono tre:

1. **Accessi non coalescenti**: i vettori u_k e v_i risiedono in memoria in posizioni lontane e imprevedibili. I thread di uno stesso warp accedono quindi a indirizzi distanti, costringendo la GPU a effettuare molteplici transazioni di memoria indipendenti e riducendo drasticamente l'efficienza del kernel;

2. **Race condition massivi**: centinaia di thread possono tentare di aggiornare simultaneamente lo stesso item (specialmente se l'item è molto popolare), causando conflitti di scrittura e comportamenti non deterministici;

3. **Alto costo delle atomic operation**: per garantire la correttezza degli aggiornamenti concorrenti, è necessario ricorrere a operazioni atomiche (come `atomicAdd`). Queste impongono che un solo thread alla volta possa modificare un dato indirizzo di memoria. Quando molti thread competono per lo stesso parametro, le atomic serializzano gli accessi: i thread vengono messi in attesa e l'intero warp rimane in stallo finché l'operazione non è completata, compromettendo gran parte del parallelismo disponibile.

1.5.2 Strategie per lo SGD su GPU

La parallelizzazione dello SGD per la matrix factorization su GPU richiede tecniche specifiche per mitigare i problemi principali. Le tecniche più comuni includono:

- **AtomicAdd per gli aggiornamenti**: utilizzato nelle implementazioni lock-free più semplici, permette a molti thread di aggiornare gli stessi parametri senza lock espliciti. Funziona bene solo in presenza di bassi livelli di conflitto [7], ma le prestazioni degradano rapidamente se item o utenti sono molto popolari.

- **Mini-batch SGD**: accumulare più gradienti prima dell'aggiornamento riduce il numero di scritture atomiche e migliora la coalescenza degli accessi [6]. Generalmente, l'uso di batch più grandi migliora il data reuse e riduce il traffico verso la global memory.

- **Partitioning per utenti o item**: dividere la matrice R in blocchi disgiunti, assegnando ciascun blocco a thread block o SM differenti, elimina i conflitti all'interno dello stesso blocco e riduce drasticamente l'uso di atomic operation. Questa tecnica è impiegata in sistemi avanzati come *cuMF_SGD* [7] e nei metodi block-based più recenti [8].

- **Caching in shared memory**: spostare nella shared memory gli item o gli utenti più popolari consente di ridurre l'accesso alla memoria globale, migliorando la località e la coalescenza [6]. Una gestione gerarchica della memoria (shared, registers, global) rappresenta uno dei fattori chiave per l'efficienza di MF su GPU.

Nessun singolo articolo garantisce che tutte e quattro le strategie siano la "ricetta magica": i lavori di successo tipicamente ne combinano alcune, scegliendo in base al dataset, alla memoria GPU, alla sparsità, etc.

1.6 Conclusioni

Lo *Stochastic Gradient Descent* rimane uno degli algoritmi più efficaci per la fattorizzazione delle matrici di rating, grazie alla sua semplicità, scalabilità teorica e capacità di gestire dataset di dimensioni enormi. Tuttavia, la sua parallelizzazione introduce sfide non banali: i conflitti sugli aggiornamenti, gli accessi irregolari alla memoria e i vincoli imposti dall'hardware rendono necessario ripensare profondamente l'implementazione rispetto alla versione sequenziale.
Le tecniche parallele su CPU, come *Distributed SGD* e *Hogwild!*, mostrano che è possibile ottenere accelerazioni significative aggirando o ignorando i conflitti, purché siano soddisfatte determinate condizioni di indipendenza dei dati o di sufficiente sparsità. Nei sistemi distribuiti, l'adozione di modelli asincroni come l'ASGD conferma che la convergenza può essere mantenuta anche in presenza di ritardi e parametri non aggiornati in modo sincrono.
Sul fronte GPU, il quadro è più complesso: nonostante il parallelismo massivo offerto dall'hardware, la natura irregolare dell'accesso ai fattori utente e prodotto rende difficile ottenere i guadagni teorici. I metodi più moderni adottano una combinazione di tecniche — partizionamento, caching, batch, ottimizzazioni della memoria — per mitigare i colli di bottiglia legati alla coalescenza degli accessi, ai conflitti e alle atomic operation.
In conclusione, la parallelizzazione dello SGD nei sistemi di raccomandazione non è un problema risolto in modo definitivo: ogni piattaforma (CPU multicore, cluster distribuiti, GPU) richiede strategie differenti, e nessuna soluzione risulta universalmente ottimale.

Riferimenti bibliografici

[1] Christopher R Aberger. «Recommender: An analysis of collaborative filtering techniques». In: *Personal and Ubiquitous Computing Journal* 5 (2014).

[2] David C Anastasiu et al. «Big data and recommender systems». In: (2016).

[3] Rainer Gemulla et al. «Large-scale matrix factorization with distributed stochastic gradient descent». In: *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2011, pp. 69–77.

[4] Benjamin Recht et al. «Hogwild!: A lock-free approach to parallelizing stochastic gradient descent». In: *Advances in neural information processing systems* 24 (2011).

[5] Jeffrey Dean et al. «Large scale distributed deep networks». In: *Advances in neural information processing systems* 25 (2012).

[6] Wei Tan et al. «Matrix factorization on gpus with memory optimization and approximate computing». In: *Proceedings of the 47th International Conference on Parallel Processing*. 2018, pp. 1–10.

[7] Peng Xie, Wang Tan et al. «Parallelized Stochastic Gradient Descent for Matrix Factorization on GPUs». In: *Proceedings of the 2017 IEEE International Conference on Big Data*. 2017.

[8] P. Bhavana e V. Padmanabhan. «GPU accelerated matrix factorization of large scale data using block based approach». In: *arXiv preprint arXiv:2304.13724* (2023).