Sparse Matrix Transposition for GPUs

Massimiliano Incudini - VR433300
Michele Penzo - VR439232

***Sommario*—L'obiettivo principale di questo progetto è stato quello di implementare alcune metodologie proposte per effettuare *Sparse Matrix Transposition* su *Gpu*. Sono stati analizzati alcuni algoritmi, descritti in sezione IV, partendo dall'algoritmo seriale, passando a cuSPARSE per finire con l'implementazione degli algoritmi descritti in [1]. Infine vengono esposti i risultati e tratte le conclusioni.**

I. Introduzione

Sempre più applicazioni computazionali in ambito scientifico necessitano di algoritmi che compiano operazioni che si possano applicare su matrici sparse. Si parla di semplici operazioni di algebra lineare, di moltiplicazione o di calcolo della trasposta come in questo caso.
Il problema analizzato, quello della trasposizione di matrici, si presta bene al calcolo parallelo per aumentarne l'efficienza. Verranno quindi mostrate le basi per la rappresentazione, i problemi riscontrati durante lo sviluppo e analizzati alcuni algoritmi per il calcolo su *Gpu*.

II. Rappresentazione delle matrici

Una matrice sparsa non è altro che una matrice i cui valori sono per la maggior parte uguali a zero. La matrice in formato classico necessita di una quantità di memoria minima di mxn elementi, ma essendo l'obiettivo quello di lavorare su matrici sparse non è stato necessario e utile memorizzare la matrice in formato denso.
Per rappresentare in modo efficace le matrici sparse senza troppo utilizzo di memoria sono state quindi introdotte ed utilizzate delle forme di rappresentazione matriciale che permettono il salvataggio di dati utilizzando quantitativi di memoria inferiori.
Di seguito vengono spiegate le due metodologie da noi utilizzate.

A. Csr

Il *compressed sparse row* è una rappresentazione di una matrice M basata su tre array monodimensionali, che rispettivamente contengono:

1) *V*: i valori *nnz*,
2) *COL_INDEX*: gli indici delle colonne dove si trovano gli elementi *nnz*,
3) *ROW_INDEX*: ha un elemento per ogni riga della matrice e rappresenta l'indice in V dove comincia la riga data.

I primi due array sono di dimensione *nnz*, mentre il terzo array è al massimo di dimensione m.

B. Csc

Questa metodologia per la rappresentazione è simile alla sopra citata *Csr*, a differenza che i valori vengono letti prima per colonna. Di conseguenza, un indice di riga viene memorizzato per ogni valore e i puntatori di colonna vengono memorizzati.

C. Da Csr a Csc

Per il problema della trasposta di matrice è stato quindi utile introdurre entrambe le rappresentazioni. Infatti, ogni algoritmo descritto in sezione IV, necessita di sei array per effettuare il calcolo della trasposta e ritornare l'output in modo corretto. Abbiamo quindi:

- in input il formato *Csr*: csrRowPtr, csrColIdx, csrVal;
- in output il formato *Csc*: cscColPtr, cscRowIdx, cscVal.

In base a come vengono create le matrici, se in modo casuale oppure se lette da file, vengono effettutate delle operazioni preliminari descritte dalla procedure in sezione V che portano ad ottenere gli array in input e in output nel formato corretto per effettuarne il controllo di correttezza.

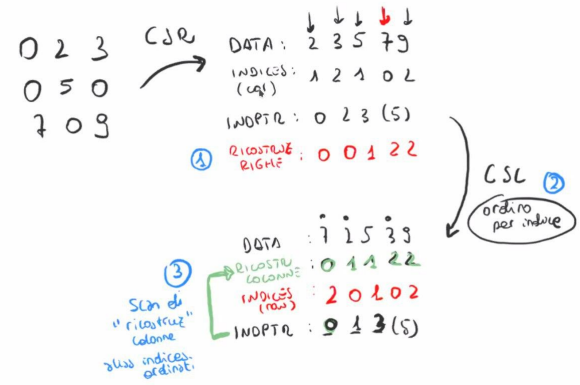

Figura 1. Esempio di trasformazione da formato Csr a Csc.

III. Struttura dell'implementazione

L'implementazione è stata sviluppata utilizzando come supporto il tool *Git*. È stata creata una repository, descritta nella sezione successiva, per poter controllare in modo efficiente lo svilupparsi del progetto. Tutti i test sono stati effettutati sulle macchine locali e sul server Cuda01.

A. Struttura delle directory

La struttura delle direcotry è rispecchiata nel seguente schema:

```
root
├── doc
│   └── report_aa.*
├── code
│   ├── matrices
│   │   └── *.mtx
│   ├── include
│   │   ├── matrix.hh
│   │   ├── procedures.hh
│   │   ├── transposer.hh
│   │   ├── Timer.*
│   │   └── utilities.hh
│   ├── src
│   │   ├── all cuda procedures..
│   │   ├── transposer.cu
│   │   └── main.cu
│   ├── test
│   │   └── test_main.cu
│   └── Makefile
└── README.md
```

Il file `matrix.hh` è composto da due classi con i relativi campi e metodi che si occupa della:

1) costruzione della matrice sparsa in formato Csr: *class SparseMatrix*,
2) costruzione della matrice densa: *class FullMatrix*.

Il file `procedures.hh` contiene tutte le procedure descritte in sezione V ed è diviso in due `namespace`. Essi riferiscono all'implementazione effettuta, ovvero *cuda* o *reference*(seriale).
È presente inoltre il file `transposers.hh` che funge da "wrapper" per le implementazioni quattro descritte nella prossima sezione.
Infine abbiamo il timer e un file che contiene tutte le funzioni utilizzate utili per debug, stampa, allocazione e deallocazione, generazione dei valori random, controllo errori e molto altro.

All'interno della direcotry *src* abbiamo tutte le implementazioni delle procedure cuda utilizzate nel progetto, oltre che al file `transposers.cu` che contiene le implementazioni descritte in IV.
Infine è presente il `main.cu`, che si occupa di eseguire tutte le metodologie implementate.

È presente inoltre un'ultima directory con relativo file che si occupa della fase di test delle singole componenti.

B. Test delle componenti

Per le singole componenti(scan, sort, index to pointers ...) è stato implementato un'altro eseguibile. Lanciando il comando *make test* viene eseguito l'applicativo che effettuta il test prima per piccole istanze e poi per grandi istanze. Cosi facendo tutte le componenti vengono testate con diversi valori.
La singola componente del programma viene quindi eseguita sia con la sua implementazione seriale, sia con quella in parallelo. Cosi facendo è stato più semplice testare le singole componenti e successivamente, dopo aver effettuato dei test complessi ed averli superati a parte, è risultato più semplice unire il tutto per ottenere l'implementazione finale.

C. Applicativo finale

Tramite il comando *make run* viene eseguito l'applicativo finale. Questo esegue per un numero di iterazioni le metodologie implementate. All'interno del tag `run` nel *Makefile* viene eseguito varie volte l'eseguibile, ogni volta con valori diversi. Il file eseguibile può essere eseguito passando:

- un valore (*file.mtx*): esegue le metodolgie su una matrice sparsa caricata da file,
- tre valori (*m n nnz*): prende tre valori e genera casualmente la matrice sparsa,
- nessun valore: matrice generate casualmente con valori fissi.

Alla fine le tempistiche vengono concatenate all'interno del file *timing_analysis.csv* per effettuarne una migliore lettura.

IV. Metodologie analizzate

In questa sezione vengono spiegate ed evidenziate le differenze tra le varie metodologie analizzate.

A. Trasposta seriale

La prima metodologia descritta è quella seriale. Sempre a partire dalla rappresentazione in formato *csr* della matrice iniziale l'algoritmo crea un array di elementi, dove per ogni colonna della matrice analizzata conta quanti elementi **nnz** ci sono. Possiamo definire questo array come un istogramma delle frequenze degli elementi delle colonne. Viene quindi applicato un algoritmo seriale di *prefix_sum* su questo array, che conterrà ora i valori corretti di **cscColPtr**. Infine gli indici di riga e i valori nel nuovo formato *csc* vengono sistemati.
Questa implementazione servirà come base sulla quale verranno eseguiti i controlli degli algoritmi successivamente implementati.

B. Nvidia cuSPARSE

Questo toolkit è implementato all'interno nelle librerie NVIDIA CUDA runtime. Le routine delle librerie vengono utilizzate per le operazioni tra vettori e matrici che sono rappresentate tramite diversi formati. Inoltre mette a disposione operazioni che permettono la conversione attraverso diverse rappresentazioni di matrici, ed inoltre la compressione in formato *csr* che è una delle più usate quando si vuole rappresentare matrici sparse in modo efficiente.
Il codice è stato sviluppato basandosi su due versioni di cuSPARSE a causa delle Gpu utilizzate. In fase di compilazione viene quindi controllata la versione usata: 9 o 10.
Nel caso in cui la versione usata sia la 10 vengono svolti alcuni ulteriori passi, come l'allocazione dello spazio necessario oltre all'allocazione del buffer per il calcolo della trasposta. Per quanto riguarda la versione 9 invece questi passi non sono

necessari.
Infine viene chiamata la procedura che effettua il calcolo della trasposta. Nel caso in cui la versione di cuSPARSE sia la 10 viene richiesto come ulteriore parametro l'algoritmo da utilizzare.
Dopo essere state eseguite entrambe ritornano i valori ottenuti in formato *csc*.

C. ScanTrans

D. MergeTrans

V. Procedure

A. Index to pointers

B. Pointers to index

C. Merge

1) Segmented Sort:

2) Merge big:

3) Merge small:

D. Scan

E. Sort

VI. Benchmark

Per la fase di test dell'implementazione sono state utilizzate principalmente due tipologie di matrici.

A. Matrici sparse casuali

A partire da dei valori fissi di *m, n, nnz* è stata creata una matrice in formato Csr utilizzando come elementi dei valori generati in modo casuale compresi tra un minimo e un massimo a scelta. Come per i valori casuali, è possibile cambiare le dimensioni della matrice e passare dei nuovi valori all'eseguibile come descritto in sezione III-C.

B. Dataset utilizzato

In [1] è stato utilizzato un dataset di matrici sparse rappresentate in diversi formati. È stato fatto lo stesso anche nel nostro progetto, sono stati scaricati alcuni esempi presi da [2], e su questi sono stati eseguiti gli algoritmi implementati. Gli esempi tratti hanno varie dimensioni e il valore relativo all'elemento *nnz* è solitamente un numero decimale. In sezione VII vegono mostrati gli esempi utilizzati e i rispettivi valori.

VII. Risultati

VIII. Conclusioni

Riferimenti bibliografici

[1] K. H. W.-C. F. Hao Wang, Weifeng Liu, "Parallel transposition of sparse data structures," *ICS '16*, 2016.

[2] Y. H. Timothy A. Davis, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software*, 2011. [Online]. Available: https://sparse.tamu.edu/