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 applicabili 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 l'esecuzione in maniera più efficiente e veloce. 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. Formato Csr

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

1) *V*: i valori non zero (*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. Formato Csc

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

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 dare l'output nella tipologia corretta. 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.

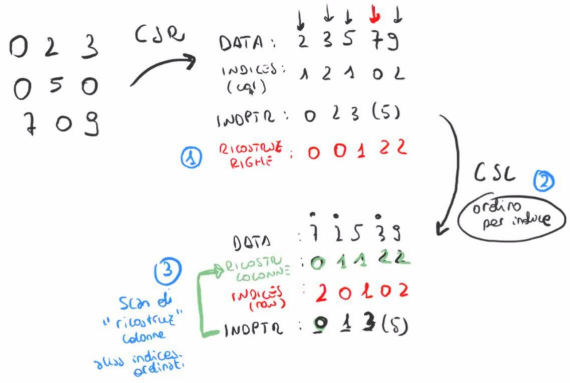

Figura 1. Esempio di trasformazione da formato Csr a Csc.

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.

Link repo: github.com/michelepenzo/architetture-avanzate

A. Struttura delle directory

La struttura delle directory è rispecchiata nel seguente schema:

```
root
├── doc
│   └── report_aa.*
├── code
│   ├── matrices
│   │   └── *.mtx
│   ├── include
│   │   ├── matrix.hh
│   │   ├── merge_step.hh
│   │   ├── procedures.hh
│   │   ├── transposers.hh
│   │   ├── Timer.*
│   │   └── utilities.hh
│   ├── src
│   │   ├── all cuda procedures..
│   │   ├── transposer.cu
│   │   └── main.cu
│   ├── test
│   │   └── test_main.cu
│   ├── Makefile
│   └── timing_analysis.csv
└── 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 quattro implementazioni descritte nella prossima sezione.
Un'altro file è il `merge_step.hh` che contiene tutte le implementazioni del *segmented merge descritto* in V-E.
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 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. Tramite questa modalità 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 i tre valori classici per la creazione della matrice e genera casualmente la matrice sparsa delle dimensioni richieste,
- 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 ottiene i puntatori alle colonne (formato csc) a partire dagli indici di colonna (formato csr). Viene quindi applicato un algoritmo seriale di *prefix_sum* su questo array, per ottenere 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 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. Supporta 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 partendo dalla guida [2] ed è diviso in 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 per l'esecuzione di cuSparse 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

L'algoritmo considerato prevede di effettuare la trasposta di matrici basandosi sul concetto di scan. Partendo sempre dal presupposto di avere in input una matrice in formato *Csr*, vengono costruiti due array ausiliari:

- inter: array bidimensionale di dimensione $(nthreads + 1) * n$,
- intra: array monodimensionale di dimensione massima nnz.

Ogni riga in *inter* contiene il numero di indici della colonna presi dalla thread i-esima. Mentre ogni elemento in *intra* viene utilizzato per salvare l'offeset relativo alla colonna corrispondente all'elemento nnz preso dalla thread. Dopo aver ottenuto gli istogrammi, viene applicato un *vertical scan* su inter, e una *prefix sum* solamente sull'ultima riga di inter. Infine l'algoritmo calcola l'offset assoluto relativo ad ogni elemento nnz e ritorna il tutto in formato *csc*.
Tutte le procedure utilizzate in *Scan Trans* si trovano in sezione V, e vengono eseguite nel seguente ordine:

1) pointers to index: V-A,
2) index to pointers: V-B,
3) scan: V-F,
4) reorder elements: V-C.

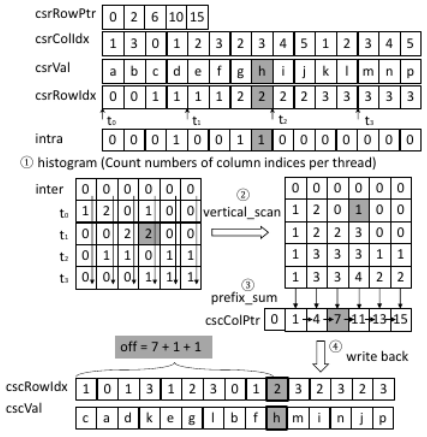

Figura 2. Scan Trans, esempio utilizzato in [1].

D. MergeTrans

L'algoritmo considerato prevede due passi importanti: *sort* e *merge*. Inizialmente sono stati creati gli indici di riga a partire dai puntatori delle colonne e su questi ultimi è stato fatto un sort su piccole porzioni di array, mantenendo quindi i vari blocchi disordinati tra di loro ma con gli elementi ordinati. Successivamente è stato utilizzato il merge ricorsivo partendo dai blocchi più piccoli e unendoli in blocchi sempre più grandi. Per funzionare questo processo necessita dell'utilizzo di due buffer di memoria che contengono gli elementi appena ordinati. Infine dai puntatori delle colonne vengono estraxtti gli indici e viene fatta la scan che ritorna il risultato in formato *csc*.
Anche in questo caso le procedure utilizzate si trovano in sezione V, e sono ordinatamente eseguite come segue:

1) pointers to index: V-A,
2) segmented sort: V-D,
3) segmented merge: V-E,
4) index to pointers: V-B,
5) scan: V-F.

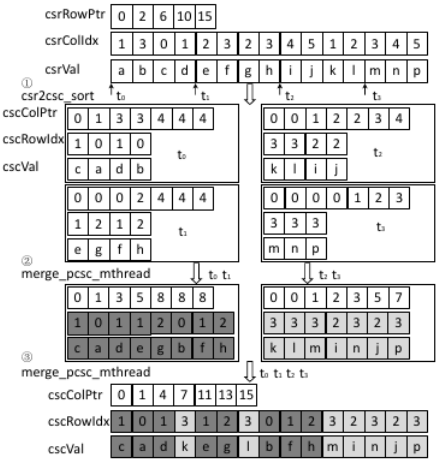

Figura 3. Merge Trans, esempio utilizzato in [1].

V. Procedure Cuda

In questa sezione vengono descritte le procedure Cuda principali che sono state utilizzate nelle implementazioni di *Scan Trans* e *Merge Trans*.

A. Pointers to index

Questa procedura prende in input l'array dei puntatori alle righe e crea un'array degli indici di riga. Quindi ad ogni thread viene assegnato un valore di *csrRowPtr* e per ognuno di questi calcola inizio e fine e crea il nuovo array.

B. Index to pointers

Lo scopo di questa procedura è quello di trasfore l'array *csrColIdx* in *colPtr*. Esegue quindi il passo (1) e (2) in figura 2:

1) *parallel histogram*: una thread per blocco (dimensione fissa) si occupa della costruzione degli array multidimensionali *inter* e *intra*. Quindi ogni blocco di thread prende un valore e data la posizione all'interno di *csrColIdx*

salva nella posizione corretta i nuovi valori all'interno di inter e intra.
2) *vertical scan*: per ogni blocco verticale all'interno di *inter* esegue una scan parallela e salva il risultato all'interno di una matrice chiamata *colPtr*.

C. Reorder elements

Ogni thread si occupa di riordinare un blocco di elementi di dimensione fissa. Viene calcolato l'*id* all'interno di *csrColIdx*, dopodichè calcolata la posizione all'interno di *csrColIdx*, *inter* e *intra* e viene posizionato il valore preso in *cscRowIdx* e *cscVal*. Questa è l'ultima procedura che viene utilizzata in *Scan Trans*, infatti si può notare che restituisce due array in formato csc.

D. Segmented sort

Questa procedura, che chiama il relativo kernel, funziona passando come parametri un numero definito a priori di thread che opera su blocchi di dimensione fissa anch'essi di $len/n\_threads$, dove *len* sta per nnz ovvero la dimensione di *colIdx*.
Anche se il concetto è lo stesso la procedura viene divisa in due per semplificare il procedimento:

- *segsort*: opera solamente su un array in input. Il kernel è stato sviluppato per funzionare con la *shared memory*, quindi abbiamo un array temporaneo condiviso tra le thread. Ogni thread lavora su un elemento del blocco, prima vengono caricati i valori nella shared memory, poi viene trovata la posizione corretta all'interno dell'array non ordinato passato in input tenendo conto dei duplicati. Dopo aver trovato la posizione, il valore viene salvato nella posizione corretta all'interno della shared memory e infine salvato nel vettore di output.
- *segsort3*: il concetto è lo stesso ma opera su 3 array in input.

E. Segmented merge

Il merge è stato realizzato tenendo conto della dimensione dei blocchi da analizzare ed è stato realizzato usando i template. Nel caso in cui abbiamo blocchi di piccola dimensione questi riescono ad essere allocati in memoria condivisa, allora viene chiamato *merge small* e viene fatto il classico merge. Nel caso in cui questi fossero di dimensione maggiore e non si riesca ad allocare una quantità così alta di memoria condivisa vengono chiamate in sequenza le seguenti procedure sue due array *A* e *B*:

1) *generate splitter*: si occupa di cercare dei valori detti splitter. Quindi ogni *splitter_distance* (costante decisa a priori) prende un elemento all'interno degli array A e B, dopodichè cerca gli indici corretti degli splitter tenendo conto dei duplicati e quindi delle posizioni corrette all'interno dell'array iniziale. Infine gli splitter e gli indici di posizione vengono salvati e dati in output.
2) *merge3 step*: si occupa dell'ordinamento degli splitter e lo effettua ricorsivamente rieseguendo i passi descritti in questo elenco. Dopo aver ottenuti tutti i blocchi ordinati prosegue con il passo successivo.
3) *fix indexes*: viene quindi sitemato l'indice finale di ogni blocco di splitter. Questo valore contiene la lunghezza del blocco, quindi ogni thread si occupa di sistemare una coppia di blocchi alla volta.
4) *uniform merge*: viene effettuato il merge di porzioni di blocchi di dimensione uniforme. Viene recuperato il blocco sulla quale lavorare e anche gli estremi da processare. Gli elementi vengono caricati in memoria condivisa e viene effettuato il merge. Infine gli elementi vengono salvati in output
5) *copy last block*: nel caso in cui restasse fuori un blocco spaiato questo viene copiato nell'array finale.

F. Scan

Questa procedura è stata divisa come per il merge in base al numero di blocchi sulla quale essa deve essere eseguita:

- *scan small*: scan effettuata su blocchi di dimensione minore a quella prefissata,
- *scan even*: scan effettuata su blocchi di dimensione esattamente multipla a quella scelta,
- *scan large*: scan effettutata prima su blocchi di dimensione multilpa (scan even) ed infine scan sui restanti blocchi (scan small).

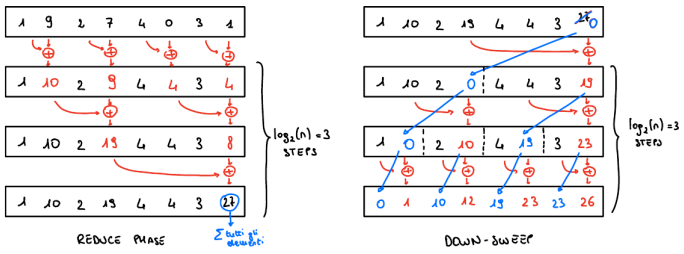

Figura 4. Parallel scan: algoritmo di Blelloch.

VI. Benchmark

Tutti i test sono stati effettuati sulle macchine locali e sul server Cuda01, quindi testati con diverse versioni di Cuda e di cuSPARSE.
Le Gpu utilizzate sono:

- macchina locale:
- server cuda01:

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 [3], 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] [Online]. Available: https://docs.nvidia.com/cuda/cusparse/index.html#abstract

[3] Y. H. Timothy A. Davis, "The university of florida sparse matrix collection," 2011. [Online]. Available: https://sparse.tamu.edu/