Sparse Matrix Transposition for GPUs

Massimiliano Incudini - VR433300
Michele Penzo - VR439232

***Sommario*—L'op 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 III, 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 III, 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 IV che portano ad ottenere gli array in input e in output nel formato corretto per effettuarne il controllo di correttezza.

III. 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.

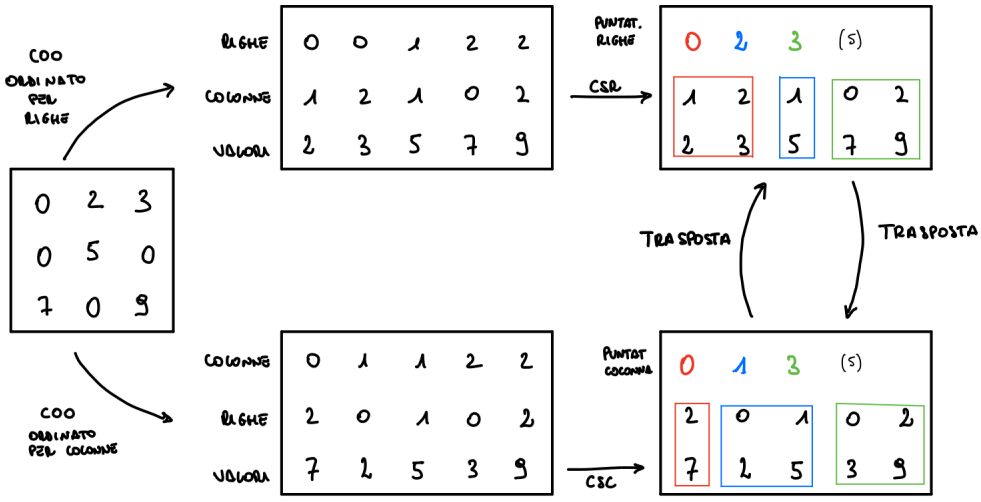

Figura 1. Trasformazione da formato esteso a CSR, oppure CSC

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 IV, e vengono eseguite nel seguente ordine:

1) pointers to index: IV-G,
2) index to pointers: IV-F,
3) scan: IV-A,
4) reorder elements.

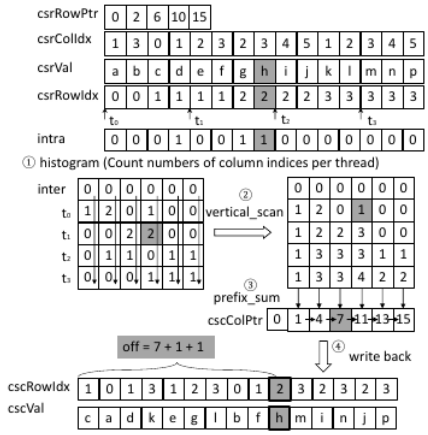

Figura 2. Scan Trans, esempio utilizzato in [1].

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 IV, e sono ordinatamente eseguite come segue:

1) pointers to index: IV-G,
2) segmented sort: IV-B,

3) segmented merge: IV-C,
4) index to pointers: IV-F,
5) scan: IV-A.

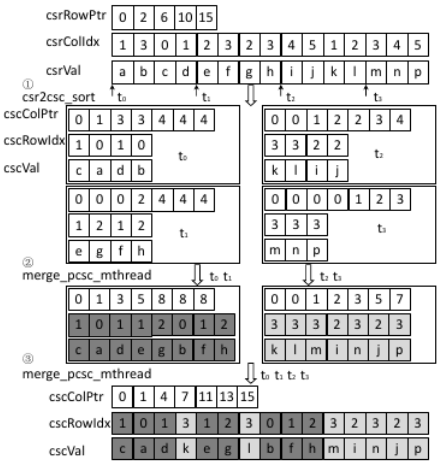

Figura 3. Merge Trans, esempio utilizzato in [1].

IV. Procedure

I due algoritmi ScanTrans e MergeTrans vengono scomposti in diversi componenti, ognuno dei quali viene valutato nelle performance e testato separatamente.

A. Scan

Questa operazione prende in input un vettore $A = (a_0, a_1, ..., a_n)$ e ritorna un vettore $B = (I, a_0, a_0 \oplus a_1, ..., a_0 \oplus a_1 \cdots \oplus a_{n-1})$ con $\oplus$ è un'operazione binaria il cui elemento identità è I. Nel nostro caso l'operazione è la somma.

L'algoritmo apparentemente sembra difficile da parallelizzare in quanto il risultato di ogni elemento dipende da tutti i gli elementi precedenti. Diverse soluzioni sono state proposte tra cui l'*algoritmo di Blelloch*. Il suo funzionamento in due fasi è illustrato in Figura 4 e dettagliatamente discusso in [3].

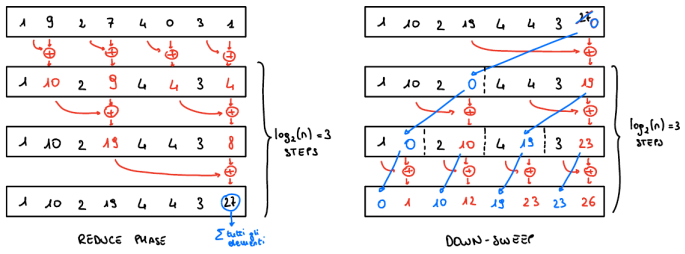

Figura 4. Algoritmo di Blelloch

L'implementazione prevede che se l'intero vettore riesce ad essere memorizzato all'interno della shared memory di N elementi, allora possiamo calcolare scan con una singola chiamata a kernel.

Nel caso questo non sia possibile, l'operazione di scan viene segmentata, applicata separatamente a blocchi di N elementi. Successivamente si mantiene un vettore di somme (vettore degli ultimi elementi del blocco), si applica ricorsivamente *scan* su esso e si sommano gli offset ottenuti all'intero vettore di partenza.

B. Segmented sort

Questa operazione prende in input un vettore di lunghezza n ed un intero BLOCK_SIZE. Il vettore viene diviso in segmenti di lunghezza BLOCK_SIZE. Gli elementi di ogni segmento vengono permutati in modo che siano ordinati stabilmente. L'intero segmento deve rientrare nella shared memory.

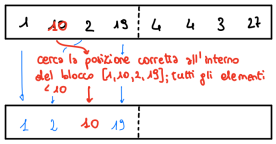

Figura 5. Segmented Sort

Una volta caricato il blocco in shared memory, la i-esima thread del blocco è incaricata di trovare la posizione corretta dell'i-esimo elemento all'interno del segmento, ed assegnarlo a tale posizione. L'algoritmo viene illustrato in Figura 5.

L'ordinamento deve essere stabile quindi la posizione dell'i-esimo elemento di valore y è dato dal numero di elementi $< y$, sommati al numero degli elementi $= y$ per indici $< i$.

La dimensione ideale del blocco pari a 128 elementi (caso interi a 32bit), ed è stata trovata empiricamente:

Thread per blocco	**Performance** (ms)
64	9999.0
128	999.5
256	99.0

Figura 6. Performance su array di $2 \cdot 10^7$ elementi

C. Merge

L'operazione di *merge* trasforma un vettore diviso in segmenti di dimensione BLOCK_SIZE nel quale gli elementi ogni segmento sono ordinati, in un vettore diviso in segmenti ordinati di dimensione 2BLOCK_SIZE, ognuno dei quali è l'unione di una coppia di blocchi contigui.

Differenziamo il caso in cui il blocco di dimensione BLOCK_SIZE rientri o meno nella shared memory.

D. Merge small

In questo caso una coppia di blocchi rientra completamente nella shared memory. In modo analogo a quanto fatto per l'operazione di *segmented sort*, abbiamo un numero di thread per blocco pari a BLOCK_SIZE nel quale l'i-esimo thread è incaricato di calcolare la posizione dell'i-esimo elemento.

In questo caso la posizione dell'i-esimo elemento del blocco di sinistra è $i + j$ con j posizione dell'elemento all'interno del blocco di destra, trovato attraverso una ricerca binaria in quanto i blocchi sono ordinati (differentemente da quanto avviene per il segmented sort). Analogamente lo stesso avviene per gli elementi del blocco di destra.

A parità di valore, gli elementi del blocco di sinistra hanno indice minore di quelli di destra.

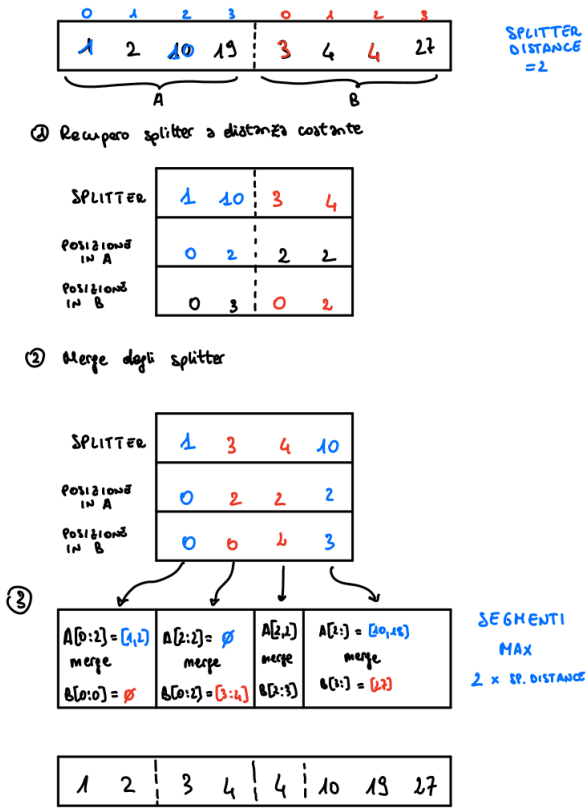

Figura 7. Merge big

E. Merge big

Questo secondo algoritmo è illustrato in Figura 7 ed originariamente preso da [4]. Applicando sempre *merge small*, oltre che rinunciare alla shared memory, ogni thread del blocco dovrebbe lavorare su più di un elemento, aumentando la complessità della procedura.

L'algoritmo proposto funziona nel seguente modo:

1) recuperiamo dal vettore degli elementi segnaposto detti *splitter* presi a distanza costante e pari ad SP_DIST; per ogni coppia di segmenti da mergiare ho una coppia di blocchi di splitter, trovo la posizione di ogni splitter all'interno del blocco di destra (A) e di sinistra (B);
2) applico *merge* ricorsivamente sugli array di splitter;
3) gli indici associati agli splitter dividono la coppia di blocchi in modo tale da poter effettuare tanti merge indipendenti quanti sono gli splitter. *Ogni merge indipendente considererà al massimo* $2SP\_DIST$ *elementi* (numero costante che scegliamo tale che rientri nella shared memory).

F. Istogramma / Index to pointers

Dato un vettore A di n elementi compresi tra 0 ed $m-1$ riceviamo un vettore di m elementi nel quale l'i-esima cella contiene la frequenza con cui il valore i è presente in A.

L'algoritmo si sviluppa in due fasi:

1) il vettore A viene diviso in N segmenti di lunghezza omogenea, ogni segmento viene processato da un blocco di thread che mantiene l'istogramma parziale;

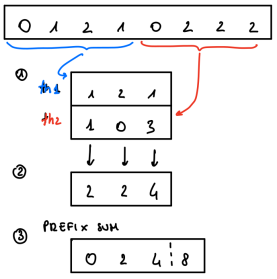

Figura 8. Index to pointers

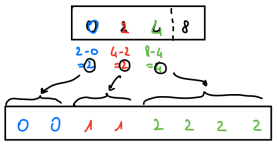

Figura 9. Pointers to index

2) gli istogrammi parziali vengono poi uniti attraverso un'operazione di prefix scan che si "sviluppa in verticale". Tale operazione può essere ottenuta:
   - trasponendo la matrice degli istogrammi parziali ed applicando N volte *prefix sum*; oppure
   - attraverso N blocchi di thread che attraversano sequenzialmente il vettore colonna degli istogrammi parziali.
3) si applica *prefix sum* sul risultato dell'operazione precedente.

L'algoritmo è illustrato in Figura 8.

G. Pointers to index

Operazione inversa di *index to pointers*. Il vettore risultante è ordinato. Può essere implementato assegnando ad ogni blocco di thread un elemento del vettore delle frequenze da espandere.

L'algoritmo è illustrato in Figura 9.

Sperimentalmente si nota che il numero ottimale di thread per blocco è 1.

V. 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 IV 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 **??**.
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 III.
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.

VI. Risultati sperimentali

Confrontiamo ora le performance delle varie implementazioni che seguono:

- seriale;
- parallela *scan trans*;
- parallela *merge trans*;
- cuSPARSE (2).

Le istanze su cui vengono eseguiti i vari algoritmi sono in parte generate in modo casuale (a partire dalle specifiche della matrice sparsa), in parte recuperate dal dataset "University of Florida sparse matrix collection" [5]. Tale dataset è stato usato per valutare le performance degli algoritmi in [1].

La macchina sul quale vengono eseguiti i vari algoritmi è equipaggiata con una scheda NVidia GeForce GTX 780 con Cuda Runtime 10.2.

I risultati sono visibili in Tabella 11.

VII. Considerazioni finali

Possiamo notare come le implementazioni citate in [1] e da noi sviluppate non siano all'altezza delle versioni di cuSPARSE

.

Contrariamente a quanto asserito in [1], nel nostro caso *Scan Trans* si comporta meglio di *Merge Trans*. Questo potrebbe essere dovuto all'implementazione da noi usata nel merge spiegato in sezione IV-C e dalle diverse ottimizzazioni utilizzate nell'implementazione del paper della quale non ne siamo a conoscenza.

Abbiamo inoltre notato come ScanTrans ottenga risultati migliori se eseguito su matrici "random" dove i valori, a differenza delle matrici in formato `.mtx`, sono interi e non decimali.

Come possibili future implementazioni per migliorare l'efficienza del progetto abbiamo pensato come il package *modern gpu* prenste su Github ci possa tornare utile. Esso mette a disposione implementazioni di alcuni componenti a noi utili per l'obiettivo finale. A partire da queste implementazioni avremmo potuto confrontare le componenti da noi sviluppate con quelle presenti per capire dove migliorare.

Nome	M	N	NNZ	Serial	ScanTrans	MergeTrans	cuSPARSE 1	cuSPARSE 2
language.mtx	399 130	399 130	1 216 334	55.75	114.71	197.05	122.02	18.36
webbase-1M.mtx	1 000 005	1 000 005	3 105 536	138.84	278.55	520.9	149.81	51
rajat21.mtx	411 676	411 676	1 893 370	77.97	147.64	306.56	127.31	26.77
ASIC_680k.mtx	682 862	682 862	3 871 773	154.37	265.47	844.76	155.31	56.74
memchip.mtx	2 707 524	2 707 524	14 810 202	594.18	994.62	2328.67	298.93	188.07
cant.mtx	62 451	62 451	2 034 917	73.83	114.4	248.49	127.26	31.11
FullChip.mtx	2 987 012	2 987 012	26 621 990	997.37	1543.02	9481.42	454.53	328.32
stomach.mtx	213 360	213 360	3 021 648	110.84	164.3	387.78	139.57	40.26
web-Google.mtx	916 428	916 428	5 105 039	399.63	382.37	3327.56	170.59	73.25
random	100 000	100 000	10 000 000	898.99	475.17	3056.12	210.57	130.25
random	100 000	100 000	10 000 000	902.23	475.72	3060.25	208.22	133.22
random	150 000	200 000	5 000 000	523.66	263.9	1351.83	161.9	72.7
random	150 000	200 000	5 000 000	527.15	262.88	1353.93	165.19	74.98
random	500 000	500 000	10 000 000	1380.96	532.76	2853.55	227.33	141.18

Figura 10. Risultati sperimentali - M, N, NNZ rispettivamente numero di righe, di colonne, di elementi non nulli della matrice. I tempi sono in ms.

Nome	M	N	NNZ	Serial	ScanTrans	MergeTrans	cuSPARSE 1	cuSPARSE 2
language.mtx	399 130	399 130	1 216 334	1.00	0.49	0.28	0.46	3.04
webbase-1M.mtx	1 000 005	1 000 005	3 105 536	1.00	0.5	0.27	0.93	2.72
rajat21.mtx	411 676	411 676	1 893 370	1.00	0.53	0.25	0.61	2.91
ASIC_680k.mtx	682 862	682 862	3 871 773	1.00	0.58	0.18	0.99	2.72
memchip.mtx	2 707 524	2 707 524	14 810 202	1.00	0.6	0.26	1.99	3.16
cant.mtx	62 451	62 451	2 034 917	1.00	0.65	0.3	0.58	2.37
FullChip.mtx	2 987 012	2 987 012	26 621 990	1.00	0.65	0.11	2.19	3.04
stomach.mtx	213 360	213 360	3 021 648	1.00	0.67	0.29	0.79	2.75
web-Google.mtx	916 428	916 428	5 105 039	1.00	1.05	0.12	2.34	5.46
random	100 000	100 000	10 000 000	1.00	1.89	0.29	4.27	6.9
random	100 000	100 000	10 000 000	1.00	1.9	0.29	4.33	6.77
random	150 000	200 000	5 000 000	1.00	1.98	0.39	3.23	7.2
random	150 000	200 000	5 000 000	1.00	2.01	0.39	3.19	7.03
random	500 000	500 000	10 000 000	1.00	2.59	0.48	6.07	9.78

Figura 11. Risultati sperimentali - Speedup

RIFERIMENTI BIBLIOGRAFICI

[1] H. Wang, W. Liu, and K. Hou, "Parallel transposition of sparse data structures," *ICS '16*, 2016.

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

[3] M. Harris and J. D. Owens, "Chapter 39. parallel prefix sum (scan) with cuda." [Online]. Available: https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda

[4] "Bad to have one big merge." [Online]. Available: https://www.youtube.com/watch?v=wx8LJIdJ_K8

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