Sparse Matrix Transposition for GPUs

Massimiliano Incudini VR433300
Michele Penzo VR439232

***Sommario*—Il progetto implementa diversi algoritmi paralleli per calcola la *trasposta* di una matrice *sparsa*, estendendo [1] riscrivendo il software per GPU NVidia attraverso il linguaggio CUDA. I risultati ottenuti vengono confrontati con le tempistiche della stessa operazione esposta dalla libreria cuSPARSE, parte dell'SDK di NVidia. Infine suggeriamo alcune modifiche per migliorare ulteriormente le performance degli algoritmi da noi implementati.**

I. Introduzione e motivazioni

Diverse applicazioni in ambito scientifico utilizzano le matrice sparse per rappresentare in modo compatto ed efficiente matrici e grafi.

La operazioni fondamentali per poter lavorare su tali strutture sono la moltiplicazione e la *trasposta*. In questo lavoro abbiamo ci siamo concentrati su questa seconda operazione. La trasposta è una componente fondamentale di diversi algoritmi come descritto in [1].

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 viene definita sparsa quando la maggior parte dei suoi valori sono nulli. Non esiste una definizione precisa, se consideriamo però una matrice di dimensioni $m \times n$ possiamo definire sparsa una matrice il cui numero di elementi non nulli nnz è più vicina ad $\max\{m, n\}$ che ad $m \times n$.

Tutti i formati di matrice sparsa permettono di memorizzare la matrice in modi molto più efficiente dal punto di vista dello storage. Alcuni formati permettono una più veloce modifica (es: *formati COO*) altri invece un più efficiente accesso al dato (es. *formati CSR, CSC*).

Una panoramica dei formati è presente in Figura 1.

A. Formato COO

Il formato COO (*COOrdinate*) rappresenta la matrice di dimensioni $m \times n$ ed nnz elementi non nulli attraverso tre vettori di lunghezza nnz:

- `coo_val`: contiene i valori non nulli;
- `coo_row_idx`: contiene gli indici di riga dei valori non nulli;
- `coo_col_idx`: contiene gli indici di colonna dei valori non nulli.

Per un accesso efficiente al dato occorre mantenere i vettori ordinati *per indice di riga* oppure per *indice di colonna*.

B. Formato CSR

Il formato CSR (*Compressed Sparse Row*) rappresenta la matrice di dimensioni $m \times n$ ed nnz elementi non nulli attraverso un vettore di lunghezza $m + 1$ e due vettori di lunghezza nnz. Può essere efficientemente ottenuto partendo dal formato *COO ordinato per righe*.

1) `csr_row_ptr`: ottenuto processando il vettore `coo_row_idx` attraverso la funzione *istogramma* che calcola la frequenza di elementi per riga, e successivamente *scan*. Il risultato che ottieniamo è che la cella $\texttt{csr\_row\_ptr}[i]$ punta al primo elemento della riga i-esima negli altri vettori, e che $R = \texttt{csr\_row\_ptr}[i+1] - \texttt{csr\_row\_ptr}[i]$ è il numero di elementi presenti nella i-esima riga.
2) `csr_col_idx`: corrisponde a `coo_col_idx`;
3) `csr_val`: corrisponde ad `coo_val`;

C. Formato CSC

Il formato CSC (*Compressed Sparse Column*) rappresenta la matrice in modo simile al formato CSR. A differenza di quest'ultimo, il formato CSC viene ottenuto a partire dalla matrice in formato *COO ordinato per colonne* e l'operazione di istogramma e scan vengono applicati alle colonne invece che alle righe.

Il vettore di puntatori `csc_col_ptr` ha lunghezza $n + 1$ a differenza di `csr_row_ptr` che ha lunghezza $m + 1$.

D. Equivalenza trasposta ↔ csr-to-csc

Risolvere il problema della trasposta in formato CSR/CSC è equivalente ad effettuare un cambio di formato da CSR a CSC (o viceversa).

III. Metodologie analizzate

Gli algoritmi paralleli che calcolano la trasposta di una matrice sparsa descritti in [1] vengono confrontati nelle tempistiche rispetto all'algoritmo seriale e rispetto alle funzioni di libreria cuSPARSE fornite da NVidia come parte della loro piattaforma Cuda SDK.

Per convenzione la matrice è in formato *csr*. I nomi delle componenti della matrice sono chiamati `csr_row_ptr` (lunghezza $\mathrm{m} + 1$) per il vettore dei puntatori ad inizio riga, `csr_col_idx` (lunghezza nnz) per il vettore degli indici di colonna e `csr_val` (lunghezza nnz) per il vettore dei valori.

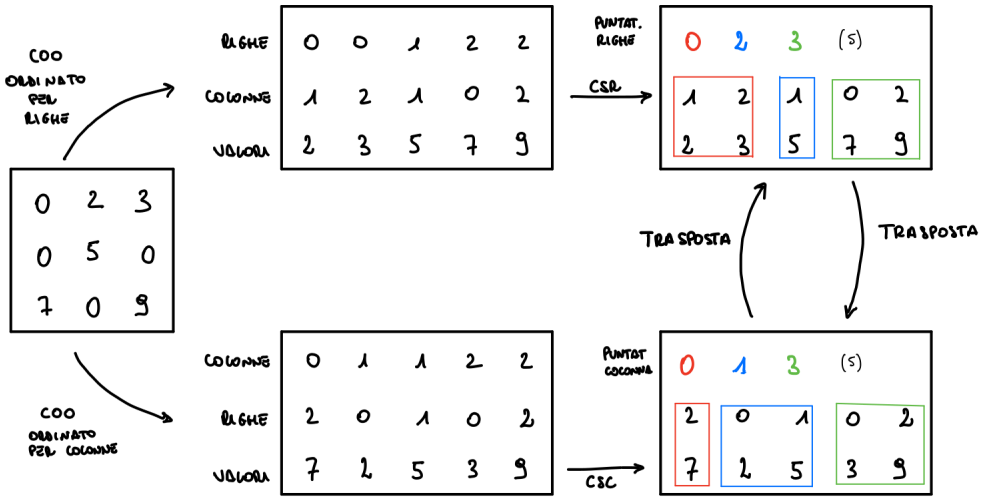

Figura 1. Trasformazione da formato esteso a CSR, oppure CSC

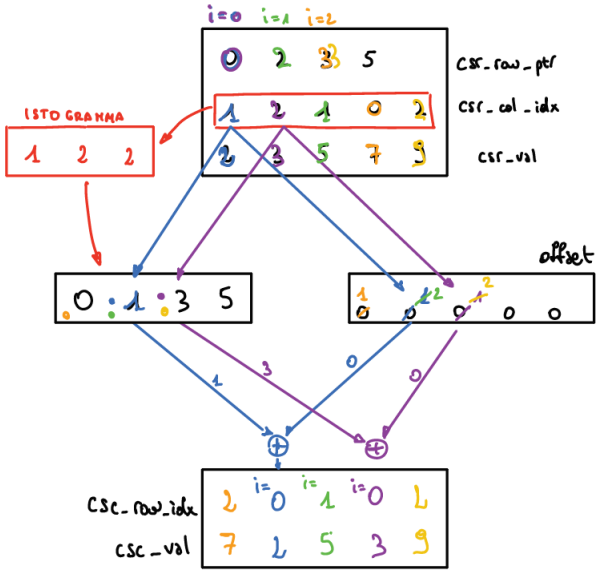

Figura 2. Algoritmo seriale

A. *Trasposta seriale*

L'algoritmo si sviluppa nel seguente modo:

1) si applica la funzione *istogramma* `csr_col_idx` che calcola le occorrenze di ogni colonna, applicando *scan* si ottiene il vettore `csc_col_ptr` (lunghezza $\mathrm{n}+1$) che conterrà i puntatori agli elementi di inizio riga trasposta;
2) allochiamo il vettore `csc_row_idx` contente gli indici di riga (lunghezza nnz);
3) allochiamo il vettore `csc_val` contente gli elementi non nulli (lunghezza nnz);
4) per ogni riga $i \in [0, m)$ processiamo gli elementi corrispondenti all'i-esima riga, legati alle posizioni $j \in [\texttt{csr\_row\_ptr}[i], \texttt{csr\_row\_ptr}[i+1])$
   - la locazione loc del j-esimo elemento ordinata per righe è $\texttt{csc\_col\_ptr}[j] + \mathrm{offset}[j]$ dove $\mathrm{offset}[j]$ è un contatore incrementato ogni volta che aggiungiamo un elemento della colonna $\texttt{csc\_col\_ptr}[j]$;
   - l'indice di riga dell'elemento loc-esimo è i;
   - il valore dell'elemento loc-esimo è $\texttt{csr\_val}[j]$.

L'esecuzione dell'algoritmo seriale è illustrato in Figura 2.

B. *ScanTrans*

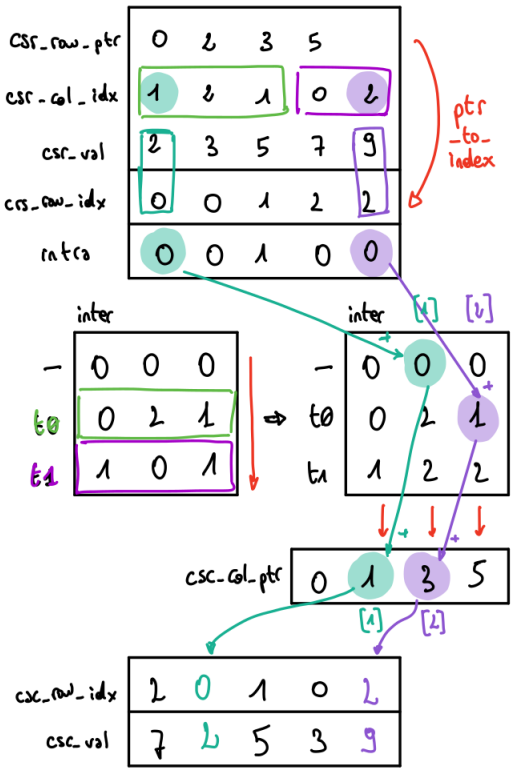

Figura 3. Algoritmo ScanTrans

L'algoritmo si sviluppa nel seguente modo:

1) il vettore `csr_row_ptr` viene espanso per ottenere il vettore `csr_row_idx` attraverso la procedura *pointers-to-indexes* che effettua l'operazione inversa dell'*istogramma*;
   ora la struttura (`csr_row_idx`, `csr_col_idx`, `csr_val`) rappresenta la matrice in formato COO ordinato per righe;
2) decidiamo un numero arbitrario di thread K che lavoreranno ciascuna su blocchi di K/nnz elementi. Allochiamo una matrice `inter` di dimensioni $(K+1) \times n$ ed un vettore `intra` di dimensione nnz. La prima manterrà nella $(i+1)$-esima riga l'istogramma parziale relativo all'i-esimo blocco. La seconda mantiene un offset di colonna di ciascun elemento;
3) i K thread eseguono l'*istogramma* ciascuna sul proprio blocco, la matrice `inter` viene riempita;
4) viene applicata l'operazione *scan* ad ogni colonna della matrice `inter`;
5) il vettore `csc_col_ptr` viene ottenuto copiando l'ultima riga di `inter` ed applicando al vettore risultante l'operazione *scan*;
6) riordiniamo gli elementi nei vettori `csr_row_idx`, `csr_val` ottenendo quindi `csc_row_idx`, `csc_val`;
   la nuova posizione dell'elemento i-esimo viene calcolata nel seguente modo:
   - $b = i/\mathrm{nnz}$;
   - $c =$ `csc_col_idx`$[i]$;
   - $l =$ `csc_col_ptr`$[c] +$ `inter`$[bn + c] +$ `inter`$[c]$;
   - `csc_val`$[l] \leftarrow$ `csr_val`$[j]$;
   - `csc_row_idx`$[l] \leftarrow$ `csr_row_idx`$[j]$.

Il funzionamento dell'algoritmo è illustrato in Figura 3.

C. MergeTrans

Il precedente algoritmo esegue nello step (6) un insieme di accessi casuali in memoria che non permettono di sfruttare in modo adeguato la shared memory.

Per mitigare questo aspetto negativo viene introdotta una variante di questo algoritmo che, se implementato correttamente, permette ai thread di lavorare sempre con un numero costante, contiguo di elementi.

L'algoritmo considerato prevede due passi importanti: *sort* e *merge*.

1) portiamo la matrice in formato COO;
2) i vettori viene segmentato in blocchi di dimensione omogenea ognuno dei quale viene ordinato per indice di colonna; ogni blocco mantiene il proprio array dei puntatori contenente l'istogramma parziale proprio del blocco;
3) i blocchi vengono uniti a coppie attraverso l'operazione di *merge* per ottenere blocchi di dimensione doppia della precedente. L'invariante di questo algoritmo è che ogni blocco mantiene la proprietà di essere ordinato per colonne ad ogni step.

Questo algoritmo permette ad ogni griglia di thread di lavorare su una porzione di memoria ben definita, che eventualmente può essere caricata in più step nella shared memory a seconda di come viene implementato l'algoritmo che effettua il *merge*.

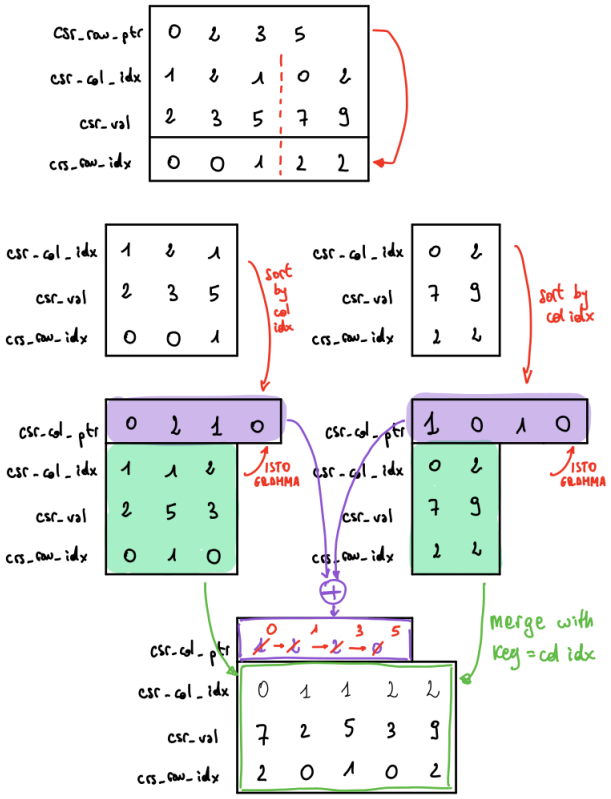

Figura 4. Algoritmo MergeTrans

Il *sort* viene applicato su blocchi di dimensione omogenea che deve essere scelta tale che l'intero blocco rientri nella shared memory.

Il funzionamento dell'algoritmo è illustrato in Figura 4.

D. NVidia cuSPARSE

I risultati temporali di tutti gli algoritmi precedentemente descritti saranno confrontati con l'implementazione di NVidia della trasposta della matrice sparsa.

La libreria cuSPARSE è parte di CUDA Toolkit, insieme di librerie utilizzate per effettuare operazioni tra vettori e matrici attraverso diversi formati.

Versioni diverse di CUDA Toolkit implementano funzioni con nomi diversi e funzionamenti diversi. In particolare per Cuda Toolkit 9.0 possiamo trasporre la matrice sparsa attraverso la procedura `cusparseScsr2csc`.

Nella versione Cuda Toolkit 10.2 viene esposto il metodo `cusparseCsr2cscEx2` che effettua la trasposta in due possibili modi a seconda se il parametro "algoritmo" è valorizzato con la costante `CUSPARSE_CSR2CSC_ALG1` oppure `CUSPARSE_CSR2CSC_ALG2`. Le due versioni differiscono per le tempistiche e per il consumo di memoria (maggior memoria necessaria per la seconda implementazione). Le tempistiche sono riportate in Tabella **??**.

La libreria è dettagliatamente documentata in [2]. A seconda della versione di *CUDA* dobbiamo modificare lo script di compilazione. In particolare il *Makefile* del progetto permette di compilare per due architetture specifiche, una con CUDA 9 e l'altra CUDA 10/11.

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 5 e dettagliatamente discusso in [3].

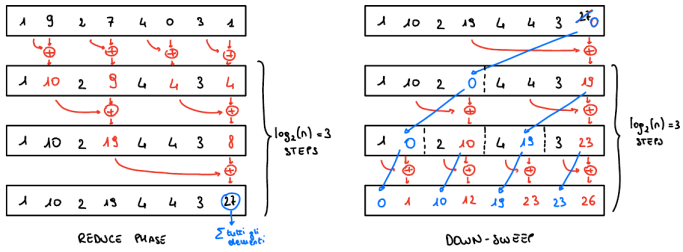

Figura 5. Algoritmo di Blelloch

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.

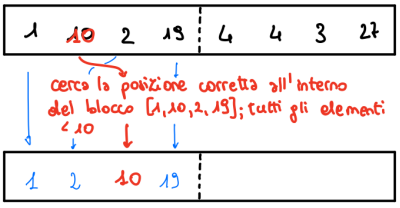

Figura 6. Segmented Sort

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 6.

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)
128	2135.58
256	2139.62
512	2156.61

Figura 7. 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 di ogni segmento sono ordinati, in un vettore diviso in segmenti ordinati di dimensione $2 *$ BLOCK_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.

E. Merge big

Questo secondo algoritmo è illustrato in Figura 8 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 unire ho una coppia di blocchi di splitter, trovo quindi 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

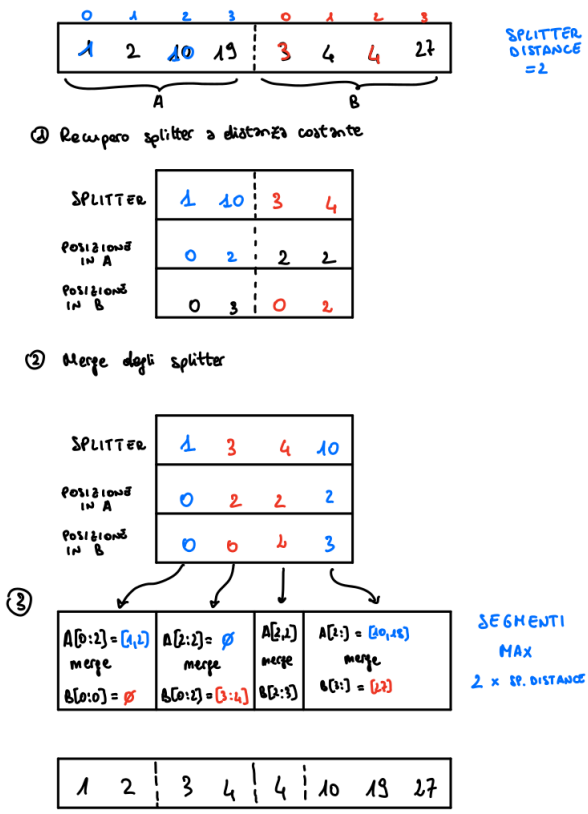

Figura 8. Merge big

indipendenti da quanti sono gli splitter. *Ogni merge indipendente considererà al massimo* $2 * SP\_DIST$ *elementi* (numero costante che scegliamo tale che rientri nella shared memory).

La dimensione ideale del blocco pari a 256 elementi (caso interi a 32bit), ed è stata trovata empiricamente:

Thread per blocco	**Performance** (ms)
128	708.87
256	701.54
512	715.71

Figura 9. Performance su array di $2 \cdot 10^7$ elementi

Mentre il valore migliore per la distanza degli splitter (SP_DIST) è di 128.

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;
2) gli istogrammi parziali vengono poi uniti attraverso un'operazione di prefix scan che si effettua "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.

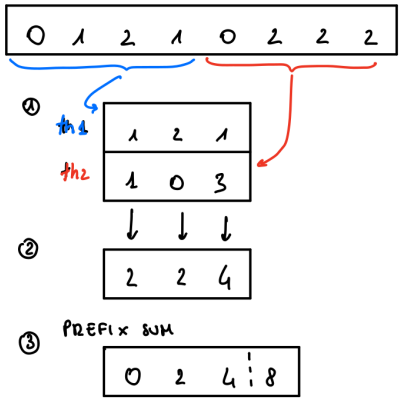

Figura 10. Index to pointers

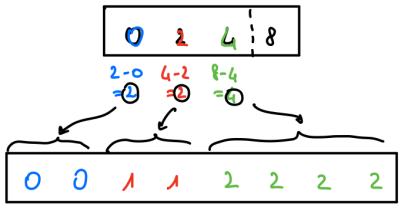

Figura 12. Pointers to index

L'algoritmo è illustrato in Figura 10.

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

Thread per blocco	**Performance** (ms)
1	1016.87
32	1013.65
64	1024.70
256	986.84

Figura 11. Performance su array di 10^6 elementi

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 12.

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

```
root
├── README.md
├── doc
└── code
    ├── matrices
    ├── include
    │   ├── matrix.hh
    │   ├── merge_step.hh
    │   ├── procedures.hh
    │   ├── transposers.hh
    │   ├── Timer.*
    │   └── utilities.hh
    ├── src
    │   ├── ...
    │   ├── transposer.cu
    │   └── main.cu
    ├── test
    │   ├── ...
    │   ├── tester.hh
    │   └── test_main.cu
    ├── Makefile
    └── timing_analysis.csv
```

Figura 14. Struttura delle directory del progetto

Thread per blocco	**Performance** (ms)
1	2328.88
16	2375.74
32	2322.75
1024	3326.13

Figura 13. Performance su array di $2 \cdot 10^7$ elementi

V. Struttura dell'implementazione

L'intera implementazione è scaricabile attraverso *git* dalla repository https://github.com/michelepenzo/architetture-avanzate.

La struttura delle directory del progetto è presente in Figura 14.

La sottodirectory *doc* contiene questo stesso documento in formato *pdf* ed i rispettivi sorgenti *tex*.

La sottodirectory *code* contiene i sorgenti dell'applicativo principale e di quello secondario di test.

Lo scopo del primo è generare un file *csv* contente le tempistiche e gli speedup di ogni algoritmo applicato sulle istanze di matrici in input generate casualmente oppure lette da file *mtx* (*market matrix*, una rappresentazione della matrice sparsa in formato COO attraverso file di testo).

Lo scopo del secondo applicativo è testare il corretto funzionamento di ogni componente del progetto. In particolare, sottopongo le stesse istanze di array o matrici (a seconda del componente che sto testando) sia alla funzione che ne implementa l'algoritmo parallelo, sia alla funzione che ne implementa l'algoritmo seriale. Mi aspetto che i risultati siano uguali per tutte le istanze.

In particolare:

- il file `include/matrix.hh` contiene le classi *FullMatrix* e *SparseMatrix* che si occupano di allocare nella memoria host lo spazio necessario a contenere la matrice date le sue specifiche (m, n, nnz), sia come matrice estesa sia in formato *csr*. Inoltre contiene i metodi per inizializzare la matrice in modo casuale e per passare da un formato all'altro;
- il file `include/utils.hh` contiene i metodi di utilità quali le funzioni di stampa e di allocazione e deallocazione della memoria device;
- i file `include/procedures.hh` e `merge_step.hh` contengono le dichiarazioni delle procedure descritte nella Sezione IV. La maggior parte delle definizioni sono presenti nella sottodirectory `src`, nel caso del metodo *merge_step* la definizione è scritta direttamente nell'header. Questa scelta è conveniente in quanto tale funzione è definita rispetto ad un tipo generico, se la definizione fosse stata riportata nei file cpp avremmo dovuto indicare esplicitamente i tipi concreti per il quale vogliamo rendere disponibile il nostro metodo ([5]);
- il file `include/transposers.hh` e rispettivo sorgente `src/transposers.cu` contengono le dichiarazioni e definizioni dei metodi che effettuano la trasposta: seriale, parallela con ScanTrans e MergeTrans ed infine da libreria cuSPARSE con entrambi i possibili algoritmi;
- i file `Timer.*` contengono una classe di utilità *timer* che permette di cronometrare il tempo che occorre per eseguire un dato pezzo di codice;
- il file `src/main.cu` contiene l'applicativo principale che chiama i diversi metodi di trasposta sulla matrice fornita in input, ne cronometra le tempistiche e le stampa in output;
- il file `test/tester.hh` contiene la classe astratta *tester* che espone un metodo *test_many_instances* che chiama un metodo astratto *test* un numero arbitrario di volte, passandogli in input un intero che rappresenta il numero dell'istanza attuale che può essere usato per decidere la dimensione dell'istanza di test. Attualmente le istanze testate vanno da 1 a 20'000, poi da 20'000 a 20'000'000 con step $\times 1.5$;
- i file `test/tester_*.hh` si occupano di testare un singolo componente, contengono ciascuna uno o più classi concrete che estendono la classe astratta *tester*;
- il file `test/test_main.cu` contiene l'applicativo di test che alloca oggetti delle varie classi tester, li avvia e ne stampa gli eventuali errori a video.

VI. Avvio degli applicativi

L'applicativo principale può essere avviato con tre modalità diverse:

- senza parametri, genera una matrice $500'000 \times 500'000$ con $10'000'000$ elementi non nulli, ne valuta le tempistiche con i diversi algoritmi ritornando la media su un numero arbitrario di esecuzioni;
- con tre parametri interi m, n, nnz, genera una matrice avente le dimensioni ricevute in input e procede alla valutazione delle tempistiche come sopra;
- con un parametro stringa `filename`, legge da file una matrice che deve essere nel formato *mtx market matrix*.

Avviando l'applicativo principale attraverso il Makefile con *make run* viene avviato molteplici volte l'applicativo principale, ogni volta con un'istanza di matrice diversa in input, con lo scopo di popolare un documento *timing_analysis.csv* contenente le tempistiche medie su diversi input.

L'applicativo di test di avvia con una sola modalità equivalentemente avviando il nome dell'applicativo senza parametro oppure attraverso il Makefile con *make test*. Su *stdout* viene stampato "no" se almeno un test ha mostrato anomalie, "ok" altrimenti.

VII. Risultati sperimentali

Confrontiamo ora le performance delle varie implementazioni che seguono:

- seriale;
- parallela *scan trans*;
- parallela *merge trans*;
- cuSPARSE (entrambi gli algoritmi).

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" [6]. 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 **??** (tempistiche) e in Tabella **??** (speedup).

Analizzando i risultati notiamo che per istanze di dimensioni notevoli l'implementazione ScanTrans raggiunge speedup fino a $\times 2.5$ rispetto all'algoritmo seriale, mentre le due implementazioni fornite da cuSPARSE raggiungono $\times 6.0$ e $\times 9.8$ di speedup.

L'implementazione MergeTrans risulta sempre meno efficiente della versione seriale dell'algoritmo. I possibili problemi relativi a questa implementazione sono discussi in Sezione VIII.

VIII. Considerazioni finali

Contrariamente a quanto asserito in [1], nel nostro caso ScanTrans si comporta meglio di MergeTrans. Proponiamo alcune idee per migliorare il risultato:

- utilizzare un algoritmo più efficiente per realizzare il merge quale il *Merge Path* ([7]);
- la procedure da noi descritte potrennero essere sostituite con quelle fornite nella libreria *modern gpu* presente su Github; inoltre seguendo la descrizione delle implementazioni presenti in [8] ci avrebbe aiutato nella fase di ingegnerizzazione.
- la procedura *istogramma* (`index_to_pointers`) può essere migliorata nel seguente modo: attualmente una griglia di un unico thread calcola l'istogramma parziale sulla sua porzione di vettore; si potrebbero introdurre tante thread ognuna che legge l'intera porzione di blocco ma si occupa di processare gli elementi appartenenti ad una sola (o ad un piccolo insieme) di valori. Questo permetterebbe anche di caricare gli elementi in shared memory, eventualmente una porzione di blocco per volta.

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] "Explicit template instantiation - when is it used?" [Online]. Available: https://stackoverflow.com/questions/2351148/explicit-template-instantiation-when-is-it-used

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

[7] D. A. B. Oded Green, Robert McColl, "Gpu merge path - a gpu merging algorithm." [Online]. Available: https://web.cs.ucdavis.edu/~amenta/f15/GPUmp.pdf

[8] [Online]. Available: https://github.com/moderngpu/moderngpu

,,,,

,,,,,,,,