Algoritmo di Trasposizione di Matrici Sparse per GPU

Elena Ramon (VR445643) e Nicola Serlonghi (VR445270)

***Sommario*—Questo documento ha come scopo quello di presentare una possibile implementazione in CUDA dell'algoritmo ScanTrans presentato nell'articolo Parallel Transposition of Sparse Data Structures [1] e il suo confronto con una versione seriale e un'implementazione di Nvidia.**

I. Introduzione

Oggigiorno molte applicazioni scientifiche richiedono il calcolo della trasposta di una matrice. Questo processo, per quanto possa essere semplice, nel caso di matrici sparse produce un elevato tempo di esecuzione ed un elevato utilizzo della memoria, nonostante la scarsità dei dati rilevanti contenuti nella matrice stessa, infatti si definisce matrice sparsa una matrice i cui valori sono quasi tutti uguali a zero [2]. Negli anni sono stati sviluppati diversi algoritmi per eseguire in modo più veloce operazioni fondamentali su questo tipo di matrici, come ad esempio la trasposizione.

L'articolo Parallel Transposition of Sparse Data Structures [1], su cui ci siamo basati per l'implementazione in CUDA, presenta due approcci per eseguire la trasposizione parallela di matrici sparse in modo efficiente su CPU, ScanTrans e MergeTrans. La versione da noi implementata è ScanTrans. Abbiamo optato per questa scelta in quanto gli autori della ricerca lo indicavano come l'algoritmo con prestazioni peggiori tra i due proposti ed eravamo interessati a capire, rispetto alla versione seriale (sempre proposta nell'articolo) e la versione implementata da Nvidia [3], quale posizione avrebbe occupato lo speedup dell'implementazione su GPU, cercando di mantenere il più possibile la struttura stessa dell'algoritmo.

Nel progetto sono state implementate tre versioni, una seriale, indicata nell'algoritmo, una parallela su GPU ideata da Nvidia e una parallela su GPU ottenuta modificando, dove possibile, la versione parallela su CPU di ScanTrans.

Come verrà illustrato nella conslusione, i nostri risultati hanno mostrato che la versione migliore tra quelle implementate, a livello di speedup, sia quella di Nvidia (il secondo algoritmo), mentre la nostra versione di ScanTrans si trova al terzo posto.

II. Backgorund

Prima di iniziare a spiegare come è strutturato l'algoritmo da noi implementato è necessario spiegare come queste matrici vengano memorizzate al fine di ridurre la quantità di memoria richiesta.

Come detto in precedenza molti valori di una matrice sparsa sono pari a zero, di conseguenza l'idea di base è quella di memorizzare i soli elementi rilevanti. Sia nnz il numero di elementi diversi da zero contenuti nella matrice, m il numero di righe della matrice e n il numero di colonne, vengono utilizzati tre array:

- csrRowPtr: di dimensione $m + 1$, memorizza i puntatori di inizio e di fine degli elementi diversi da zero delle righe;
- csrColIdx: di dimensione nnz, memorizza l'indice di colonna dei valori diversi da zero;
- csrVal: di dimensione nnz, memorizza tutti i valori diversi da zero.

che nella corrispettiva versione trasposta risultano essere:

- cscColPtr: di dimensione $n + 1$, memorizza i puntatori di inizio e di fine degli elementi diversi da zero delle colonne;
- cscRowIdx: di dimensione nnz, memorizza l'indice di riga dei valori diversi da zero;
- cscVal: di dimensione nnz, memorizza tutti i valori diversi da zero.

III. Versioni implementate

A. Seriale

La prima versione che viene presentata è quella seriale, la quale si svolge in tre fasi:

Fase 1 viene inizializzato cscColPtr in modo tale da contenere il numero di elementi diversi da zero in ogni colonna.

Fase 2 viene eseguita la somma cumulativa (prefix sum) su cscColPtr.

Fase 3 cscColPtr viene usato per calcolare l'offset degli elementi diversi da zero rispetto alla loro posizione corrente. Avviene quindi la loro trasposizione.

B. Nvidia

Per quanto riguarda la versione di Nvidia è stato necessario eseguire la chiamata delle due funzioni previste dalla documentazione [3]. La prima funzione ritorna la quantità aggiuntiva di memoria che deve essere allocata e successivamente passata alla seconda funzione che esegue la trasposizione. Nvidia indicati due possibili algoritmi per eseguire la trasposizione:

- CUSPARSE_CSR2CSC_ALG1: richiede un ulteriore quantità di memoria proporzionale a nnz;
- CUSPARSE_CSR2CSC_ALG2: richiede un ulteriore quantità di memoria proporzionale a m, risulta essere più performante su matrici regolari.

Nel progetto vengono eseguiti entrambi, abbiamo preso questa decisione in modo da avere un ulteriore parametro per poter effettuare un confronto sugli speedup.

C. ScanTrans

La terza versione si basa sullo pseudo-codice presente nell'articolo relativo all'algoritmo ScanTrans, il quale è stato pensato per essere eseguito su CPU e parallelizzato attraverso OpenMP [4].

L'idea di base di questo algoritmo è di spartire in modo equo il numero di elementi diversi da zero tra i threads. Utilizza tre utleriori array di supporto:

- inter: di dimensione $(nthreads + 1) * n$, dove nthreads è il numero di thread in esecuzione, memorizza il numero di volte in cui un indice di colonna viene osservato;
- intra: di dimensione nnz, memorizza l'offset nella colonna dell'elemento diverso da zero;
- csrRowIdx: di dimensione nnz, memorizza gli indici di riga degli elementi diversi da zero.

e si sviluppa in cinque fasi:

Fase 1 genera l'indice di riga degli elementi diversi da zero, i quali vengono posti in csrRowIdx;
Fase 2 ogni thread prende gli indici di colonna da csrColIdx e aggiorna intra e inter;
Fase 3 viene eseguita la somma cumulativa in colonna su inter in modo tale da avere nell'ultima riga il numero totale di elementi diversi da zero in ogni colonna;
Fase 4 viene eseguita la somma cumulativa dell'ultima riga di inter che successivamente viene inserita in cscColPtr;
Fase 5 viene calcolato l'offset assoluto di ogni elemento diverso da zero e vengono riempiti cscVal e cscRowIdx.

L'implementazione proposta si sviluppa nella chiamata di quattro kernel:

Kernel 1 include la Fase 1 e la Fase 2 della versione per CPU;
Kernel 2 si occupa della Fase 3;
Kernel 3 viene eseguita la prefix sum;
Kernel 4 si occupa della Fase 5.

Abbiamo definito più kernel in quanto un thread durante l'esecuzione complessiva dell'algoritmo accede alle variabili in punti differenti. Ad esempio nel Kernel 1 ogni thread lavora su una determinata riga di inter a cui accede solo lui, nel kernel successivo invece ogni thread lavora su una determinata colonna di inter a cui accede solo lui, di conseguenza queste due operazioni non possono essere messe nello stesso kernel senza una sincronizzazione esplicita di tutti i blocchi di thread, altrimenti una volta ultimata la scansione per righe un thread passerebbe alla scansione in colonna senza che tutti i dati siano stati aggiornati.

L'articolo presenta un algoritmo per implementare in modo parallelo ed efficiente prefix sum su CPU, la versione da noi presentata invece utilizza un algoritmo implementato da Nvidia [5] e quindi ottimizzato per GPU.

Per scegliere il numero di threads con cui far partire ogni kernel abbiamo deciso di utilizzare una funzione [6] che calcola, in base alla dimensione massima tra il numero di righe e il numero di colonne della matrice, la dimensione e il numero ottimale dei blocchi per massimizzare l'occupancy della GPU.

Numero di multiprocessori	12
Numero massimo di threads per multiprocessore	2048
Numero massimo di threads per blocco	1024
Dimensione dei warp	32
Memoria	3GB

Tabella I: Caratteristiche GPU Server Università degli Studi di Verona

Nome	#M/N	#NNZ	ALGO1	ALGO2	ScanTrans
ASIC_680k	682862	3871773	14.6x	14.4x	low memory
cage14	1505785	27130349	18.2x	11.8x	low memory
circuit5M	5558326	59524291	13.0x	12.7x	low memory
eu-2005	862664	19235140	11.6x	17.1x	low memory
flickr	820878	9837214	28.3x	18.8x	low memory
FullChip	2987012	26621990	14.1x	13.9x	low memory
language	399130	1216334	15.0x	14.3x	low memory
memchip	2707524	14810202	12.9x	13.4x	low memory
para-4	153226	5326228	12.1x	8.8x	low memory
rajat21	411676	1893370	11.3x	16.1x	low memory
rajat29	643994	4866270	11.2x	13.1x	low memory
sme3Dc	42930	3148656	11.8x	9.0x	low memory
Stanford_Berkley	683446	7583376	16.7x	17.5x	low memory
stomach	213360	3021648	9.6x	12.1x	low memory
torso1	116158	8516500	16.3x	19.5x	low memory
transient	178866	961790	16.2x	18.9x	low memory
venkat01	64242	1717792	11.2x	15.3x	low memory
webbase-1M	1000005	3105536	16.0x	19.9x	low memory
web-Google	916428	5105039	62.8x	33.7x	low memory
wiki-Talk	2394385	5021410	82.3x	35.0x	low memory

Tabella II: Benchmark articolo con speedup - esecuzione su GPU proprietaria

IV. Analisi delle performance

Il principale problema che abbiamo riscontrato riguarda la memoria richiesta per l'esecuzione dell'algoritmo. Infatti la GPU su cui abbiamo sviluppato il nostro progetto (si veda tabella I) ha 3GB di memoria, il che significa che non è stato possibile testarlo sulle matrici utilizzate come benchmark nell'articolo perché troppo grandi (si veda tabella II).

Abbiamo eseguito dei test su una GPU differente (si veda tabella III), ma ancora la memoria richiesta era troppo grande. Abbiamo quindi cercato altre matrici di dimensione minore [7] riuscendo ad ottenere speedup e non uscendo dalla memoria disponibile (si veda tabella IV)

V. Conclusione

Questo articolo mostra una possibile implementazione dell'algoritmo ScanTrans di trasposizione di una matrice sparsa

Numero di multiprocessori	10
Numero massimo di threads per multiprocessore	2048
Numero massimo di threads per blocco	1024
Dimensione dei warp	32
Memoria	6GB

Tabella III: Caratteristiche GPU proprietaria

Nome	#M/N	#NNZ	ALGO1	ALGO2	ScanTrans
human_gene1	22283	24669643	21.1x	18.4x	2.2x
psmigr_3	3140	543162	9.0x	13.4x	1.3x
bcsstk16	4884	290378	10.1x	20.4x	1.4x
psmigr_2	3140	540022	9.1x	16.1x	1.2x

Tabella IV: Benchmark con speedup - esecuzione su GPU proprietaria

su GPU, confrontato con una versione seriale e due versioni parallele su GPU di Nvidia.

La nostra idea era quella di non modificare l'algoritmo presentato nell'articolo, ma cercare di adattarlo il più possibile all'architettura GPU. Abbiamo osservato che lo speedup dipende non solo dalla dimensione della matrice, ma anche dal numero di elementi diversi da zero, come si vede dalla tabella IV.

Il problema dell'utilizzo eccessivo di memoria, causato dai tre array di supporto utilizzati da ScanTrans, potrebbe essere risolto diminuendo il numero di thread e di conseguenza la dimensione di inter, ma a questo punto si verificherebbe un peggioramento delle performance, dato che un singolo thread dovrebbe svolgere più lavoro. La suddivisione di questi array in dimensioni più piccole non è un'alternativa implementabile in quanto l'accesso ad essi è casuale.

In media su tutti i test eseguiti da ogni algoritmo la versione di Nvidia ALGO1 risulta essere la migliore con una media di 18.9x, al secondo posto c'è la seconda versione di Nvidia ALGO2 con una media di 16.9x ed infine al terzo posto con 1.5x la versione di ScanTrans su GPU, la quale è stata testata solo su quattro matrici di medie dimensioni.

RIFERIMENTI BIBLIOGRAFICI

[1] H. Wang, W. Liu, K. Hou, and W. chun Feng, "Parallel transposition of sparse data structures." [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/2925426.2926291

[2] "Matrice sparsa." [Online]. Available: https://it.wikipedia.org/wiki/Matrice_sparsa

[3] "Traposizione parallela di matrici sparse - versione nvidia." [Online]. Available: https://docs.nvidia.com/cuda/cusparse/index.html#csr2cscEx2

[4] "Openmp." [Online]. Available: https://www.openmp.org/

[5] M. Harris, "Parallel prefix sum (scan) with cuda." [Online]. Available: http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/scan/doc/scan.pdf

[6] "Cuda occupancy." [Online]. Available: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__HIGHLEVEL.html#group__CUDART__HIGHLEVEL_1gee5334618ed4bb0871e4559a77643fc1

[7] "Suitesparse matrix collection." [Online]. Available: https://sparse.tamu.edu