Algoritmo di Trasposizione di Matrici Sparse per GPU

Elena Ramon e Nicola Serlonghi

***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 accessi in memoria su GPU sono particolarmente lenti e tra i due algoritmi proposti questo risulta essere quello con il minor numero di accessi e secondo noi il più promettente.

In questo documento vengono messe a confronto quattro implementazioni, una seriale, indicata nell'articolo, due algoritmi proposti da Nvidia e una parallela su GPU ottenuta modificando, dove possibile, la versione parallela su CPU di ScanTrans.

Come verrà illustrato nella conclusione, i nostri risultati hanno mostrato che la versione migliore tra quelle implementate, a livello di speedup, sia quella di Nvidia, 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:

Histogram include la Fase 1 e la Fase 2 della versione per CPU;
Vertical Scan si occupa della Fase 3;
Prefix Sum viene eseguita la prefix sum;
Write Back 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 in Histogram 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.

IV. Analisi delle performance

Abbiamo testato l'intero progetto su due architetture differenti, sia in termini di processore che di scheda grafica, la prima, a cui ci riferiremo con l'espressione "Architettura-server", viene presentata nelle tabelle I per quanto riguarda la GPU e II per quanto riguarda la CPU; la seconda, a cui ci riferiremo con l'espressione "Architettura-proprietaria", viene presentata nelle tabelle III per la GPU e IV per la CPU.

Lo speedup di ogni versione parallela presentata è stato ottenuto da un confronto rispetto alla versione seriale proposta nell'articolo. Abbiamo calcolato i tempi di esecuzione delle quattro implementazioni in tre modalità:

1) tempo comprensivo di allocazioni di memoria, copia dei dati da CPU e GPU, esecuzione dei kernel, copia dei risultati da GPU a CPU e pulizia della memoria, tabelle VI e VII;
2) tempo comprensivo di allocazioni di memoria, copia dei dati da CPU e GPU ed esecuzione dei kernel, tabelle VIII e IX;
3) tempo riguardante le sole esecuzioni dei kernel, tabelle X e XI.

La tabella V riporta le matrici su cui abbiamo effettuato i test. Le prime venti sono state proposte dall'articolo di riferimento, le ultime quattro, di dimensione inferiore, le abbiamo introdotte noi a seguito del problema riguardante la quantità di memoria richiesta per l'esecuzione dell'algoritmo ScanTrans.

Gli speedup ottenuti dalla nostra implementazione sono inferiori rispetto a quelli delle versioni di Nvidia, ma comunque positivi all'aumentare della dimensione delle matrici e del numero di elementi diversi da zero. Vedendo l'andamento dei risultati ci aspettiamo un aumento dello speedup all'aumentare delle dimensioni delle matrici, ma non è stato possibile verificarlo a causa della limitata memoria disponibile su entrambe le architetture.

Il collo di bottiglia che ci rallenta rispetto alla versione di Nvidia lo abbiamo individuato nel kernel Write Back in quanto in quest'ultimo accediamo a tutte le strutture dati, è un accesso univoco che ci obbliga a lasciarle nella memoria globale.

V. Conclusione

Questo documento mostra una possibile implementazione dell'algoritmo ScanTrans di trasposizione di una matrice sparsa 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 VII.

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 nella modalità 1 da ogni algoritmo le versioni di Nvidia, ALGO1 e ALGO2 sono a pari merito con una media di 4.2x, al secondo posto con una media di 1.3x abbiamo la versione di ScanTrans su GPU.

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

APPENDICE

Modello	Nvidia GeForce GTX 780
Memoria	3 GB GDDR5
Architecture	Kepler
Bus interface	PCIe 3.0 x16
Base core clock	863 MHz
Bost core Clock	902 MHz
Memory Clock	1502 MHz
Numero di multiprocessori	12
Numero massimo di threads per multiprocessore	2048
Numero massimo di threads per blocco	1024
Dimensione dei warp	32

Tabella I: Caratteristiche GPU Server Università degli Studi di Verona

Modello	AMD Phenom II X6 1055T
Cores	6
Threads	6
Clockspeed	2.8 GHz
Turbo Speed	3.3 GHz
L1 cache	64 KB
L2 cache	512 KB
L3 cache	6144 KB

Tabella II: Caratteristiche CPU Server Università degli Studi di Verona

Modello	Nvidia GeForce GTX 1060
Memoria	6GB GDDR5
Architecture	Pascal
Bus interface	PCIe 3.0 x16
Base core Clock	1506 MHz
Bost core Clock	1709 MHz
Memory Clock	2002 MHz
Numero di multiprocessori	10
Numero massimo di threads per multiprocessore	2048
Numero massimo di threads per blocco	1024
Dimensione dei warp	32

Tabella III: Caratteristiche GPU proprietaria

Modello	Intel Core i7-4790
Cores	4
Threads	8
Clockspeed	3.6 GHz
Turbo Speed	4.0 GHz
L1 cache	128 KB
L2 cache	1 MB
L3 cache	8 MB

Tabella IV: Caratteristiche CPU propretaria

Nome	#M/N	#NNZ
ASIC_680k	682862	3871773
cage14	1505785	27130349
circuit5M	5558326	59524291
eu-2005	862664	19235140
flickr	820878	9837214
FullChip	2987012	26621990
language	399130	1216334
memchip	2707524	14810202
para-4	153226	5326228
rajat21	411676	1893370
rajat29	643994	4866270
sme3Dc	42930	3148656
Stanford_Berkley	683446	7583376
stomach	213360	3021648
torso1	116158	8516500
transient	178866	961790
venkat01	64242	1717792
webbase-1M	1000005	3105536
web-Google	916428	5105039
wiki-Talk	2394385	5021410
bcsstk16	4884	290378
human_gene1	22283	24669643
psmigr_3	3140	543162
psmigr_2	3140	540022

Tabella V: Benchmark

Nome	ALGO1	ALGO2	ScanTrans
ASIC_680k	2.1x	2.1x	low mem
cage14	2.7x	2.6x	low mem
circuit5M	2.2x	2.2x	low mem
eu-2005	1.8x	1.9x	low mem
flickr	5.7x	5.1x	low mem
FullChip	2.3x	2.3x	low mem
language	2.4x	2.4x	low mem
memchip	1.9x	1.9x	low mem
para-4	2.1x	1.9x	low mem
rajat21	1.7x	1.8x	low mem
rajat29	1.8x	1.9x	low mem
sme3Dc	2.6x	2.4x	low mem
Stanford_Berkley	2.3x	2.3x	low mem
stomach	1.5x	1.5x	low mem
torso1	2.7x	3.0x	low mem
transient	1.6x	1.7x	low mem
venkat01	1.5x	1.5x	low mem
webbase-1M	2.3x	2.4x	low mem
web-Google	12.4x	9.8x	low mem
wiki-Talk	11.9x	9.2x	low mem
bcsstk16	0.9x	0.9x	0.3x
human_gene1	4.7x	4.5x	1.6x
psmigr_3	1.4x	1.6x	0.7x
psmigr_2	1.3x	1.5x	0.6x

Tabella VI: Speedup modalità 1 su Archiettura-proprietaria

Nome	ALGO1	ALGO2	ScanTrans
ASIC_680k	3.3x	3.3x	low mem
cage14	3.7x	3.7x	low mem
circuit5M	3.0x	3.2x	low mem
eu-2005	3.0x	3.1x	low mem
flickr	7.1x	6.9x	low mem
FullChip	3.5x	3.6x	low mem
language	3.8x	3.9x	low mem
memchip	3.2x	3.4x	low mem
para-4	2.7x	3.2x	low mem
rajat21	2.9x	3.0x	low mem
rajat29	3.1x	3.1x	low mem
sme3Dc	4.6x	4.6x	low mem
Stanford_Berkley	3.2x	3.2x	low mem
stomach	2.8x	2.9x	low mem
torso1	3.9x	4.0x	low mem
transient	2.9x	3.0x	low mem
venkat01	2.5x	2.6x	low mem
webbase-1M	3.5x	3.6x	low mem
web-Google	14.1x	13.2x	low mem
wiki-Talk	10.4x	9.7x	low mem
bcsstk16	1.6x	1.8x	0.5x
human_gene1	6.5x	6.5x	2.9x
psmigr_3	2.3x	2.5x	1.0x
psmigr_2	2.3x	2.5x	1.0x

Tabella VII: Speedup modalità 1 su Archietttura-server

Nome	ALGO1	ALGO2	ScanTrans
ASIC_680k	6.9x	7.0x	low mem
cage14	7.7x	7.9x	low mem
circuit5M	6.6x	7.1x	low mem
eu-2005	6.4x	7.0x	low mem
flickr	14.8x	14.1x	low mem
FullChip	7.6x	8.0x	low mem
language	7.4x	7.8x	low mem
memchip	7.0x	7.5x	low mem
para-4	6.7x	6.8x	low mem
rajat21	5.8x	6.4x	low mem
rajat29	6.5x	6.7x	low mem
sme3Dc	8.8x	8.9x	low mem
Stanford_Berkley	6.8x	6.9x	low mem
stomach	5.5x	6.1x	low mem
torso1	8.4x	8.9x	low mem
transient	5.4x	5.9x	low mem
venkat01	5.0x	5.7x	low mem
webbase-1M	7.2x	7.7x	low mem
web-Google	28.0x	25.0x	low mem
wiki-Talk	21.7x	18.6x	low mem
bcsstk16	2.8x	3.1x	1.1x
human_gene1	13.8x	13.7x	5.9x
psmigr_3	4.3x	4.9x	1.6x
psmigr_2	4.4x	4.1x	1.6x

Tabella IX: Speedup modalità 2 su Archietttura-server

Nome	ALGO1	ALGO2	ScanTrans
ASIC_680k	3.6x	3.6x	low mem
cage14	4.8x	4.2x	low mem
circuit5M	3.8x	3.8x	low mem
eu-2005	3.2x	3.9x	low mem
flickr	9.4x	7.7x	low mem
FullChip	4.1x	4.1x	low mem
language	4.3x	4.1x	low mem
memchip	3.4x	3.5x	low mem
para-4	3.6x	3.2x	low mem
rajat21	3.0x	3.3x	low mem
rajat29	3.2x	3.5x	low mem
sme3Dc	4.2x	3.9x	low mem
Stanford_Berkley	3.1x	4.2x	low mem
stomach	2.6x	2.7x	low mem
torso1	4.0x	5.4x	low mem
transient	2.7x	2.9x	low mem
venkat01	2.5x	2.7x	low mem
webbase-1M	4.2x	4.6x	low mem
web-Google	18.8x	13.3x	low mem
wiki-Talk	20.8x	11.8x	low mem
bcsstk16	1.3x	1.6x	0.4x
human_gene1	7.9x	7.5x	1.9x
psmigr_3	2.3x	2.7x	0.9x
psmigr_2	2.3x	2.8x	0.9x

Tabella VIII: Speedup modalità 2 su Archiettura-proprietaria

Nome	ALGO1	ALGO2	ScanTrans
ASIC_680k	10.6x	10.8x	low mem
cage14	12.8x	9.3x	low mem
circuit5M	10.6x	10.4x	low mem
eu-2005	10.0x	14.1x	low mem
flickr	21.8x	14.5x	low mem
FullChip	11.1x	11.5x	low mem
language	12.6x	11.5x	low mem
memchip	9.7x	10.2x	low mem
para-4	10.2x	7.2x	low mem
rajat21	9.0x	12.7x	low mem
rajat29	9.2x	11.0x	low mem
sme3Dc	9.5x	7.8x	low mem
Stanford_Berkley	11.9x	13.4x	low mem
stomach	8.0x	9.8x	low mem
torso1	14.1x	17.7x	low mem
transient	8.7x	9.9x	low mem
venkat01	6.0x	13.7x	low mem
webbase-1M	20.2x	24.6x	low mem
web-Google	38.0x	20.7x	low mem
wiki-Talk	58.1x	24.4x	low mem
bcsstk16	5.4x	8.7x	0.7x
human_gene1	19.5x	17.2x	2.3x
psmigr_3	7.8x	14.3x	1.4x
psmigr_2	7.4x	10.9x	1.3x

Tabella X: Speedup modalità 3 su Archiettura-proprietaria

Nome	ALGO1	ALGO2	ScanTrans
ASIC_680k	25.2x	28.7x	low mem
cage14	30.0x	30.8x	low mem
circuit5M	27.4x	33.4x	low mem
eu-2005	25.4x	37.7x	low mem
flickr	49.2x	43.1x	low mem
FullChip	28.8x	37.2x	low mem
language	26.3x	30.1x	low mem
memchip	26.7x	38.7x	low mem
para-4	26.4x	26.5x	low mem
rajat21	20.6x	28.1x	low mem
rajat29	25.3x	30.4x	low mem
sme3Dc	26.9x	28.7x	low mem
Stanford_Berkley	28.4x	36.5x	low mem
stomach	21.6x	33.5x	low mem
torso1	32.1x	39.8x	low mem
transient	18.4x	23.2x	low mem
venkat01	18.1x	29.3x	low mem
webbase-1M	28.0x	36.9x	low mem
web-Google	81.9x	60.8x	low mem
wiki-Talk	79.0x	25.3x	low mem
bcsstk16	9.4x	14.4x	1.7x
human_gene1	44.0x	54.2x	9.2x
psmigr_3	11.7x	16.5x	2.2x
psmigr_2	12.0x	18.2x	2.2x

Tabella XI: Speedup modalità 3 su Archietttura-server