Prodotto tra una matrice sparsa ed un multivettore

Luca Capotombolo, Ludovico Zarrelli

1 Luglio 2023

Contents

1 Introduzione

1.1 Contesto

Il progetto prevede la realizzazione di un nucleo di calcolo per il prodotto tra una matrice sparsa ed un multivettore:

$$Y = AX$$

dove A è una matrice sparsa memorizzata nei formati *CSR* e *ELLPACK*. Il nucleo di calcolo è stato sviluppato in C ed è stato parallelizzato per sfruttare le risorse di calcolo disponibili con parallelizzazione *OpenMP* e *CUDA*.

Nel progetto sono state utilizzate le matrici nel formato *MatrixMarket*. Per determinare la tipologia di matrice rappresentata nel file Matrix Market sono state utilizzate le funzioni di I/O disponibili all'indirizzo: `https://sparse.tamu.edu/`

1.2 Matrici di test

Per quanto riguarda le matrici di test, sono state scaricate nel formato *Matrix Market* e sono state utilizzate le funzioni per l'I/O disponibili all'indirizzo: `http://math.nist.gov/MatrixMarket/`

1.3 Implementazione Seriale

I risultati ottenuti dall'esecuzione parallela SpMM sono stati confrontati direttamente con i risultati ottenuti dall'esecuzione seriale. Per quanto riguarda le implementazioni seriali, sono state sviluppate tre differenti versioni:

- Prodotto seriale utilizzando il formato CSR
- Prodotto seriale utilizzando il formato ELLPACK con *zero padding*
- Prodotto seriale utilizzando il formato ELLPACK senza *zero padding*

Per avere una misura dell'errore, abbiamo deciso di calcolare due differenti errori che ci forniscono diverse informazioni:

- *Massimo errore relativo*: mi fornisce delle informazioni su quale sia l'errore relativo più grande tra tutti i confronti elemento per elemento.
- *Errore relativo medi0*: mi fornisce delle informazioni sulla media degli errori relativi su tutti i confronti elements by elements

Questi valori vanno confrontati con un valore di tolleranza: IEEE unit roundoff per il tipo di dato double, che è $2.2204460492503131e-016$

2 Funzioni ausiliarie per il pre-processamento dei dati

La funzione *create_matrix_coo* ha il compito di leggere i dati da uno specifico file nel formato *MatrixMarket* per poi allocare e popolare la matrice nel formato *COO*. All'interno di questa funzione vengono gestite le possibili tipologie di matrici che sono supportate dal programma:

- *Simmetrica Pattern*
- *Simmetrica Reale*
- *Generale Pattern*
- *Generale Reale*

Le strutture dati che vengono utilizzate per la rappresentazione della matrice *COO* sono le seguenti:

- I: array contenente gli indici di riga degli elementi non zero
- J: array contenente gli indici di colonna degli elementi non zero
- *val*: array contenente i valori degli elementi non zero

In questo modo, l'elemento *i-esimo* di questi tre array rappresenta rispettivamente l'indice di riga, l'indice di colonna e il valore di uno specifico elemento non zero della matrice.

3 Discussione implementazione CSR

Passiamo alla discussione relativa al formato CSR. Vedremo gli aspetti riguardanti la conversione della matrice dal formato *COO* al formato *CSR* e gli aspetti che riguardano il prodotto SpMM sia con *OpenMP* che con *CUDA*.

3.1 Funzioni ausiliarie per la memorizzazione nel formato CSR

Le strutture dati che vengono utilizzate per il formato *CSR* sono le seguenti:

- *as*: è l'array dei coefficienti non zero
- *ja*: è l'array degli indici di colonna dei coefficienti non zero
- *irp*: è l'array dei puntatori all'inizio di ciascuna riga

Durante lo sviluppo del progetto, abbiamo avuto dei problemi nella conversione della matrice dal formato *COO* al formato *CSR*. Più precisamente, le nostre difficoltà erano dovute al grande tempo richiesto per eseguire la conversione per le matrici di grande dimensione.
La prima implementazione dell'algoritmo di conversione è rappresentata dalla funzione *coo_to_CSR_parallel* che si comporta bene per le matrici di piccole/medie dimensione ma ha un tempo di esecuzione elevato per la matrici di grande dimensione. Il primo step dell'algoritmo consiste nel computare il numero di non zeri per ogni riga popolando la struttura dati di appoggio *nz_per_row* che poi viene utilizzata per inizializzare la struttura dati *irp*. A questo punto, le tre strutture dati precedentemente elencate vengono popolate utilizzando un doppio ciclo *for*. Per velocizzare l'esecuzione, questo blocco di codice è stato parallelizzato cercando di bilanciare il più possibile il carico di lavoro. Per liberare la memoria precedentemente occupata, al termine dell'inizializzazione delle tre strutture dati *as, ja* e *irp* vengono deallocate le strutture dati per la rappresentazione nel formato *COO*.
Il problema della prima implementazione è rappresentato dalla presenza dei due cicli for, che comporta un numero di iterazioni pari a M * nz, dove M è il numero di righe e nz è il numero complessivo di non zeri della matrice in input. Per ridurre il tempo richiesto nella conversione dal formato *COO* al formato *CSR*, in modo da trattare anche le matrici molto grandi, abbiamo deciso di implementare una seconda versione dell'algoritmo di conversione nella funzione *coo_to_CSR_parallel_optimization*. Il vantaggio di questa seconda versione consiste nell'utilizzo di un singolo ciclo che va ad iterare su tutti i non zeri della matrice. Per gestire la concorrenza, abbiamo introdotto una minima sezione critica che, comunque, ci consente di avere dei tempi di esecuzione molto inferiori rispetto alla versione precedente.

3.2 OpenMP

Il prodotto parallelo per il formato *CSR* con parallelizzazione *OpenMP* è implementato nella funzione *parallel_product_CSR*. La funzione *compute_chunk_size* ha il compito di calcolare la dimensione del chunk che deve essere assegnato a ciascun thread. Lo spazio delle iterazioni viene suddiviso a seconda del numero di processori che sono disponibili sul device. Inoltre, per evitare il false cache sharing, il valore $\frac{Numero\_di\_iterazioni}{Numero\_di\_processori}$ è arrotondato all'intero più vicino (inferiore) multiplo di 16. L'obiettivo è quello di assegnare un blocco di righe contiguo ai threads nello spazio di iterazione in modo da evita il più possibile il problema del *False Cache Sharing*. Ad ogni thread viene associato un certo numero di righe e ogni riga è associata ad un unico thread. In questo modo, posso utilizzare una variabile privata *partial_sum* per computare le somme parziali. Abbiamo scelto la clausola static perché abbiamo assunto che ogni iterazione abbia un costo per lo più costante.

3.3 CUDA

Per quanto riguarda il prodotto parallelo per il formato *CSR* in *CUDA*, abbiamo deciso di scrivere differenti versioni del kernel. In realtà, le prime tre versioni del kernel potrebbero essere viste come la stessa versione con solamente delle modifiche minimali che le distinguono. Abbiamo deciso di riportare queste modifiche minimali differenziando così le tre versioni del kernel poiché hanno un grande impatto prestazionale. Poiché stiamo facendo calcolo ad alte prestazioni, è importante notare come queste differenze, che sembrano essere minimali, in realtà hanno un gran peso sulle prestazioni.
L'idea che c'è dietro alle prime tre versioni del kernel è quella di far sì che un singolo thread computi un singolo elemento della matrice finale *Y*. Inizialmente, viene allocata la memoria necessaria sul *CUDA Device* e vengono copiati i dati dall'host verso il device. Il numero di threads per blocco che viene utilizzato è pari a *512* (Alcune prove empiriche hanno rivelato un miglioramento delle performance per un numero di threads per blocco pari a 512). Nel kernel vengono calcolati gli indici di riga e di colonna che identificano l'elemento che deve essere calcolato dal thread. Come primo step viene calcolato l'identificativo globale del thread. Successivamente, vengono calcolati i due indici nel seguente modo:

$$row = tid \ / \ K$$
$$col = tid \ \% \ K$$

Una volta determinata la riga *row* della matrice sparsa *A* e la colonna *col* della matrice densa *X*, viene eseguito un controllo per verificare se il thread sta calcolando effettivamente un elemento della matrice. Questo controllo mi consente di gestire una griglia di blocchi che ha un numero totale di threads strettamente maggiore del numero di elementi della matrice che bisogna calcolare.
La prima versione del kernel non è ottimizzata poiché nel calcolo del singolo prodotto scalare si accede in memoria ogni volta che il thread modifica il risultato intermedio. Questi accessi continui portano a delle prestazioni molto basse. Un modo evidente per risolvere questo problema prestazionale consiste nello scrivere in memoria direttamente il risultato finale della computazione rappresentato dal valore della variabile *partial_sum*. Questa modifica permette di aumentare notevolmente le prestazioni del prodotto parallelo. A questo punto, è stata apportata un'ulteriore modifica minimale che consiste nel pre-computarsi le due variabili *start* e *end* e successivamente eseguire il ciclo. La terza versione riesce a guadagnare in termini prestazionali rispetto alla seconda versione.

Dopo aver implementato queste tre versioni del kernel abbiamo provato a distribuire la computazione richiesta per il calcolo di un singolo elemento della matrice. Questa quarta versione è stata ispirata al *CSR Vector* descritto nei vari paper. L'idea è quella di distribuire il calcolo di un elemento della matrice finale Y tra i threads di uno stesso warp. Più precisamente, ogni thread in un

warp computa un risultato parziale per uno stesso elemento y della matrice Y. Viene eseguita una riduzione parallela che coinvolge i thread di uno stesso warp sfruttando la *shared memory*. Poiché tutti i threads di un warp contribuiscono al calcolo dello stesso elemento, è sufficiente che un solo thread tra essi scriva il risultato ottenuto dalla riduzione in memoria globale. Per fare ciò, abbiamo deciso che solamente il thread con identificativo *0* all'interno del warp ha il compito di scrivere il risultato in memoria globale. Tuttavia, questa nuova versione del kernel ci ha portato ad una notevole riduzione delle prestazioni. Il motivo per cui le prestazioni si sono ridotte così drasticamente è dovuto al numero molto piccolo di non zeri per riga. Facendo un'analisi generale delle matrici di test, abbiamo potuto osservare come gran parte di esse abbiano un numero di non zeri medio che è inferiore rispetto alla dimensione del warp. Questo implica il fatto che ci siano mediamente un numero non piccolo di thread all'interno di un warp che non danno un contributo effettivo al calcolo dell'elemento.
Un altro fattore impattante è sicuramente il numero di accessi in memoria condivisa. Rispetto alle tre versione di kernel, descritte in precedenza, abbiamo lo stesso numero complessivo di accessi in memoria globale per calcolare le somme parziali, ma abbiamo un notevole numero di accessi in memoria condivisa per realizzare la riduzione parallela e la scrittura finale in memoria globale.

Nelle prime tre versioni, ogni elemento veniva computato da un singolo thread che eseguiva il prodotto scalare riga *row* per colonna *cols*. Nella versione successiva, invece di utilizzare un unico thread, abbiamo provato a distribuire la computazione tra i differenti thread di uno stesso warp. Tuttavia, abbiamo visto che assegnando un singolo warp alla computazione di un singolo elemento y della matrice finale Y si ha una grande perdita nelle prestazioni. A questo punto, abbiamo pensato di seguire un approccio intermedio che consiste nel vedere i due casi implementati finora come dei *casi estremi*. Invece di assegnare un thread per elemento o un warp per elemento abbiamo provato a scegliere un numero intermedio di thread come compromesso tra i due differenti approcci.

Più precisamente, dato un *warp* di 32 thread, abbiamo creato 16 differenti *sub-warp*, ognuno con il compito di computare un singolo elemento della matrice finale Y. Di conseguenza, ogni elemento y della matrice finale Y è computato da due differenti thread (è un valore arbitrario che abbiamo scelto dopo un'attenta analisi empirica delle prestazioni al variare del numero di threads per ogni sub-warp) che costituiscono il sub-warp. Nella versione *CSR_Vector_Sub_warp* si ragiona in termini di identificativo del sub-warp. Come primo step si determina l'identificativo globale del thread per poi computare l'identificativo globale del *sub-warp* a cui il thread appartiene. Poiché un *sub-warp* ha una dimensione pari a *2*, ogni thread che appartiene al *sub-warp* ha un proprio identificativo locale al suo interno. Più precisamente, se la dimensione è pari a *2* allora i possibili indici dei threads all'interno di un sub-warp sono *0* e *1*. Dopo aver calcolato l'indice del sub-warp è possibile determinare la riga e la colonna dell'elemento della matrice assegnato al sub-warp. Poiché il sub-warp si occupa di computare le somme parziali per un singolo elemento della matrice Y allora possiamo uti-

lizzare direttamente una sola area di memoria associata ad ogni thread. Da notare che questo non sarebbe stato possibile se il sub-warp avesse calcolato più elementi distinti della matrice finale *Y*. In questo caso, sarebbe stato necessario un meccanismo che permettesse di distinguere le somme parziali per i differenti elementi. In questo kernel viene utilizzata la *shared memory* per mantenere i risultati delle somme parziali. Più precisamente, ad ogni thread viene assegnata una specifica entry nell'array *vals* in cui mantenere i risultati delle somme parziali per il calcolo dell'elemento. Dopo aver calcolato i propri contributi, viene eseguita una riduzione parallela nella memoria condivisa. Infine, solamente il thread con indice *0* all'interno del sub-warp scriverà il risultato in memoria globale.

Come suggerito dalla traccia del progetto, abbiamo provato ad implementare anche l'algoritmo CSR-Adaptive, ideato per spMV, ma opportunamente adattato per spMM. Per prima cosa vengono individuati i blocchi di righe, in modo tale che per ogni blocco sia assegnata una quantità di memoria condivisa al più pari a local_size (Numero medio di non-zeri per riga) * sizeof(double). Ovviamente, local_size è limitato a 1024, che moltiplicato per 8, è minore di 49152, la quantità totale di memoria condivisa per blocco in bytes dichiarata nella scheda tecnica della GPU. Successivamente, viene invocato il kernel *CSR_Adaptive* con un numero di blocchi per griglia pari a (numero di blocchi di righe - 1) * K e un numero di threads per blocco pari a local_size. Essendo la memoria condivisa limitata a local_size double non ha senso assegnare piu thread, di questo valore, per blocco. La parte complessa nella "conversione" da algoritmo per spMV a spMM è stata necessità di estenderlo in modo tale da calcolare i prodotti per le restanti K - 1 colonne della matrice X. Notare che, rispetto alla versione originale. abbiamo un numero di blocchi per griglia che è K volte più grande. Infatti, l'idea è di avere un blocco che calcola un solo elemento della matrice risultante Y. La riga e la colonna da assegnare al blocco corrente sono calcolare nel seguente modo:

$row = d\_rowBlocks[\ \frac{blockIdx.x}{K}\ ]$ con d_rowBlocks array contenente l'indice della riga iniziale per ogni blocco

$$col = blockIdx.x\%K$$

CSR-Adapive è un algoritmo che decide dinamicamente se utilizzare CSR-Stream, quando il numero di righe per blocco > 1 o CSR-Vector, quando il numero di righe per blocco di righe per blocco pari a 1.
CSR-Stream si basa sulla seguente idea: ogni thread del blocco scrive in memoria condivisa la componente parziale, di sua competenza, del prodotto finale. Per esempio il thread i del blocco j calcola il prodotto tra $as[irp[row]+i]*X[(j\%K)*N+ja[irp[row]+i]$. Successivamente, M threads eseguono la riduzione scalare dalla memoria condivisa per calcolare l'output finale. In questa versione per sfruttare maggiormente i dati in cache, la matrice X è stata trasposta in modo

tale che gli elementi della colonna siano contigui in memoria.

Anche in questo caso, non soddisfatti delle prestazioni ottenute abbiamo realizzato le seguenti versione ottimizzate per CSR-Vector e CSR-Adaptive:

- *CSR_Vector_by_row*: Ogni warp è responsabile del calcolo di tutti le componenti di una riga della matrice risultante ed è suddiviso in 8 sub-warp. I threads del sub-warp collaborano per calcolare un elemento in Y, la cui riga è determinata dal warp di cui il sub-warp fa parte. Attraverso la memoria condivisa alcuni thread in un sub-warp eseguono una riduzione parallela. Abbiamo 8 sub-warp, ciascuno dei quali composto da 4 threads, del warp j responsabile della riga i che si suddividono le K colonne di X per calcolare la riga i di Y.

- *CSR_Adaptive_sub_blocks*: Ogni blocco di thread calcola tutte le componenti di una riga della matrice Y ed è suddiviso in un certo numero di sub-block. Ogni sub-block di sub_block_size threads è responsabile del calcolo di una prodotto della componente di sua competenza per la riga i nella matrice Y. I threads dello stesso sub-block collaborano tra di loro per calcolare l'output finale.

In entrambi i casi la matrice X è stata trasposta in modo tale che gli elementi della colonna siano contigui in memoria.

4 Discussione implementazione ELLPACK

Passiamo alla discussione relativa al formato ELLPACK. Vedremo gli aspetti riguardanti la conversione della matrice dal formato *COO* al formato *ELLPACK* e gli aspetti che riguardano il prodotto SpMM sia con *OpenMP* che con *CUDA*.

4.1 Funzioni ausiliarie per la memorizzazione nel formato ELLPACK

Le strutture dati che vengono utilizzate per il formato *ELLPACK* sono le seguenti:

- *values*

- *col_indices*

Similmente a quanto descritto nella sezione relativa al formato *CSR*, per rendere più efficiente la convesione della matrice dal formato *COO* al formato *ELLPACK* abbiamo implementato tre differenti versioni per l'algoritmo di conversione che si differenziano per il tempo di esecuzione.

La prima versione dell'algoritmo di conversione è stata implementata nella funzione *coo_to_ellpack_parallel()*. In questa prima versione, viene utilizzato il

padding 0x0 per il formato *ELLPACK*. Come primo step viene calcolato il massimo numero degli elementi non-zero *max_so_far* tra tutte le righe della matrice sparsa (vedi che sul codice non è cosi...strano). Dopo aver calcolato il massimo numero di non zeri per riga vengono allocate le strutture dati *values* e *col_indices*. A questo punto, si passa con il popolare i due array bidimensionali con i valori che si trovano nelle strutture dati J e *val* del formato *COO*. Più precisamente, fissata la riga *i-esima* assegnata ad un thread T, vengono identificati tutti quanti gli elementi non-zero appartenenti a quella riga e popolo le strutture dati *values* e *col_indices* con rispettivamente il valore e l'indice di colonna dell'elemento non-zero.

Con la prima versione dell'algoritmo di conversione abbiamo riscontrato dei problemi di memoria (*Out Of Memory*) durante l'esecuzione del programma sulle nostre macchine. Per poter eseguire il programma in locale abbiamo deciso di implementare un'altra versione dell'algoritmo di conversione. La seconda versione ha l'obiettivo di risparmiare memoria nella memorizzazione della matrice nel formato *ELLPACK*. Più precisamente, abbiamo deciso di non utilizzare il *padding 0x0* e di gestire le strutture dati *values* e *col_indices* non necessariamente come tipici array bidimensionali. In questa versione, abbiamo calcolati il numero degli elemento non-zero all'interno di ciascuna riga della matrice popolando il vettore *nz_per_row*.

Tuttavia, sebbene la seconda versione ci ha permesso di risparmiare memoria aggirando il problema dell'esecuzione locale, entrambe le versioni precedenti richiedono tanto tempo per la conversione dal formato *COO* al formato *ELLPACK* delle matrici di grande dimensioni. Per scavalcare quest'ultimo ostacolo, abbiamo deciso di implementare una terza versione dell'algoritmo di conversione che ci permettesse di eseguire la conversione in un tempo ragionevole per una qualsiasi matrice di test richiesta per il progetto.

La terza versione dell'algoritmo di conversione mantiene l'idea di non utilizzare lo *zero padding* per ridurre la quantità di memoria utilizzata. Il primo step consiste sempre nel computarsi il numero di non zeri per riga popolando la struttura dati *nz_per_row*. A questo punto, invece di eseguire due cicli for annidati (procedimento adottato nelle precedenti due versioni dell'algoritmo di conversione) si sfrutta un singolo ciclo for con una piccola sezione critica al suo interno. La sezione critica mi consente di evitare un ulteriore ciclo e, nel complesso, di ridurre il tempo di esecuzione per la conversione. Nelle precedenti versioni, un solo thread si occupava della singola riga e lo spazio delle iterazioni era l'insieme delle righe. Nella terza versione, viene partizionato lo spazio degli elementi in modo che gli elementi di una singola riga possano essere assegnati a thread differenti. Viene utilizzata la struttura dati *curr_idx_per_row* che mantiene le informazioni relative all'indice attuale per le varie righe. Questa informazione è necessaria da mantenere poiché più thread possono agire lavorare su una stessa riga.

4.2 OpenMP

Il prodotto parallelo per il formato *ELLPACK* con parallelizzazione *OpenMP* è implementato nelle seguenti due funzioni:

- *parallel_product_ellpack()*
- *parallel_product_ellpack_no_zero_padding()*

Nel prodotto parallelo con *ELLPACK* abbiamo deciso di utilizzare la trasposta della matrice X che viene calcolata tramite la funzione *transpose*. Per quanto riguarda la distribuzione del carico di lavoro tra i vari thread, abbiamo deciso di utilizzare la clausola *schedule* con *static*. In questo modo, le iterazioni del ciclo sono divise in parti con dimensione pari a *chunk_size* e sono staticamente assegnati ai thread. Il valore di *chunk_size* viene calcolato considerando il numero M di righe della matrice e il numero di processori *ntherad* disponibili sul dispositivo. L'idea è quella di distribuire le righe della matrice tra i vari thread che sono disponibili per l'esecuzione. Inoltre, utilizziamo anche la clausola *collapse* poiché abbiamo due differenti cicli innestati perfettamente (il ciclo che itera su M e quello che itera su K). La differenza tra le due funzioni *parallel_product_ellpack()* e *parallel_product_ellpack_no_zero_padding()* sta nella gestione delle righe poiché la versione senza *zero padding* utilizza strutture dati leggermente differenti. Infatti, nella versione che utilizza il padding il ciclo for più interno itera fino al numero massimo di non zeri per riga mentre nella versione ottimizzata si itera fino alla dimensione effettiva di quella riga che ottengo tramite la struttura dati *nz_per_row*.

4.3 CUDA

Per quanto riguarda il prodotto parallelo per il formato *ELLPACK* in *CUDA*, abbiamo deciso di implementare due differenti versioni del kernel:

- *ELLPACK_kernel*: L'idea di far sì che un singolo thread computi un singolo elemento della matrice finale Y. Anche in questo caso il numero di thread per blocco che viene utilizzato è pari a *512*. Nel kernel vengono calcolati gli indici di riga e di colonna che identificano l'elemento che deve essere calcolato dal thread:

  $$row = tid \;/\; K$$

  $$col = tid \;\%\; K$$

  Una volta determinata la riga *row* della matrice sparsa A e la colonna *col* della matrice densa X, viene eseguito un controllo per verificare se il tid del thread strettamente minore del numero di elementi della matrice che bisogna calcolare.

- *ELLPACK_Sub_warp*: Abbiamo 16 differenti *sub-warp* di 2 thread ciascuno, ognuno con il compito di computare un singolo elemento della matrice finale *Y*. Come primo step si determina l'identificativo globale del thread per poi computare l'identificativo globale del *sub-warp* a cui il thread appartiene. Poiché un *sub-warp* ha una dimensione pari a *2*, ogni thread che appartiene al *sub-warp* ha un proprio identificativo locale al suo interno. Dopo aver calcolato l'indice del sub-warp è possibile determinare la riga e la colonna dell'elemento della matrice assegnato al sub-warp. Poiché il sub-warp si occupa di computare le somme parziali per un singolo elemento della matrice Y allora possiamo utilizzare direttamente una sola area di memoria associata ad ogni thread. In questo kernel viene utilizzata la *shared memory* per mantenere i risultati delle somme parziali. Più precisamente, ad ogni thread viene assegnata una specifica entry nell'array *vals* in cui mantenere i risultati delle somme parziali per il calcolo dell'elemento. Dopo aver calcolato i propri contributi, viene eseguita una riduzione parallela nella memoria condivisa. Infine, solamente il thread con indice *0* all'interno del sub-warp scriverà il risultato in memoria globale.

5 Misurazione delle prestazioni

6 Suddivisione del lavoro

7 Istruzioni per la compilazione