Prodotto tra una matrice sparsa ed un multivettore

Luca Capotombolo, Ludovico Zarrelli

1 Luglio 2023

Contents

1 Introduzione

1.1 Presentazione del problema

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 al meglio le risorse di calcolo disponibili con parallelizzazione *OpenMP* e *CUDA*.

1.2 Matrici di test e funzioni di I/O

Nel progetto sono state utilizzate le matrici nel formato *MatrixMarket* scaricabili all'indirizzo `https://sparse.tamu.edu/`. Inoltre, per facilitare la gestione del formato MatrixMarket sono state utilizzate delle funzioni per l'I/O disponibili all'indirizzo `http://math.nist.gov/MatrixMarket/`. Ad esempio, le funzioni di I/O sono state utilizzate per determinare la tipologia di matrice rappresentata nel file in formato MatrixMarket.

1.3 Implementazione Seriale

I risultati ottenuti dall'esecuzione parallela sono stati confrontati 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*

Come verrà detto successivamente nella relazione, sono state implementate due versioni del formato ELLPACK per risolvere un problema legato all'utilizzo eccessivo della memoria.

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 corrispondente 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*

Una matrice pattern è una rappresentazione di una matrice sparsa, in cui vengono memorizzate solo le posizioni (riga e colonna) degli elementi non nulli della matrice, senza conservare i valori effettivi (si assume che sono tutti uni) di tali elementi.

Le matrici reali simmetriche contengono valori reali e soddisfano la proprietà di simmetria, cioè A(i,j) = A(j,i).

Le matrici reali non simmetriche (generali) sparse contengono valori reali e non hanno restrizioni di simmetria.

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 riscontrato dei problemi nella conversione della matrice dal formato *COO* al formato *CSR*. Più precisamente, le nostre difficoltà erano legate alla grande quantità di tempo richiesta per eseguire la conversione delle 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 passo dell'algoritmo consiste nel computare il numero di non zeri per ogni riga popolando la struttura dati di supporto *nz_per_row* che poi verrà 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, al termine dell'inizializzazione delle tre strutture dati *as, ja* e *irp* vengono deallocate le strutture dati utilizzate per la rappresentazione della matrice 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*, soprattutto per trattare le matrici che hanno una grande dimensione, abbiamo deciso di implementare una seconda versione dell'algoritmo di conversione rappresentata dalla funzione *coo_to_CSR_parallel_optimization*. Il vantaggio di questa seconda versione consiste nell'utilizzo di un singolo ciclo *for* che itera 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 rappresentato dalle M righe della matrice viene suddiviso a seconda del numero di processori che sono disponibili sul device.
Nell'analisi del prodotto parallelo con OpenMP utilizzando il formato CSR possiamo soffermarci a ragionare sui seguenti due aspetti:

- *False Cache Sharing*: Il "False Cache Sharing" è una situazione in cui due o più variabili o dati che sono logicamente indipendenti condividono la stessa linea di cache. Questo può causare un problema di prestazioni nei sistemi con gerarchie di memoria cache, poiché le operazioni su queste variabili apparentemente indipendenti possono comportare conflitti sulla stessa linea di cache, portando a un maggior numero di invalidazioni e ricariche di cache, con conseguente degradazione delle prestazioni.

Supponiamo di avere due variabili "A" e "B" in un programma multi-thread. Queste variabili non sono correlate tra loro e vengono utilizzate da thread diversi. Tuttavia, a causa della loro posizione in memoria, finiscono per condividere la stessa linea di cache. Se un thread modifica il valore di "A", la linea di cache contenente "A" verrà invalidata, causando anche l'invalidazione della linea di cache contenente "B", anche se "B" non è stato toccato da quel thread. Questo comporta un'inefficienza in termini di accesso alla cache e può causare una riduzione delle prestazioni, a causa del fatto che la nostra architettura viene inondata di transizioni distribuite. Di conseguenza, il false cache sharing si ha quando due cache controller richiedono l'uso esclusivo, per poter scrivere, di una stessa cache line.

- *Utilizzo della località*

Come si può intuire, le prestazioni del prodotto parallelo saranno molto legate a questi due aspetti. Per comprendere ciò che effettivamente accade, ricordiamo che le strutture dati su cui stiamo lavorando per il formato CSR sono le seguenti:

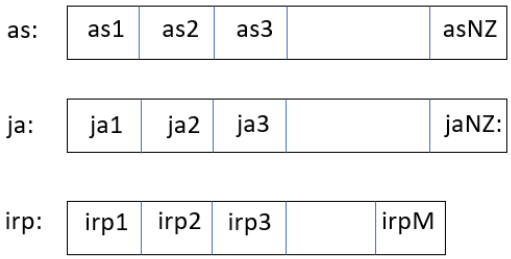


Figure 1: Strutture dati CSR

Inoltre, la funzione *create_dense_matrix()* crea una matrice densa X di dimensione N x K strutturata nel seguente modo:

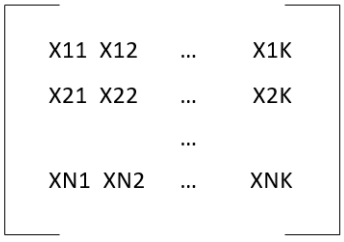


Figure 2: Matrice Densa

Prima di continuare, facciamo la seguente osservazione:
La memorizzazione di una matrice può avvenire in due modi principali: in for-

mato 2D (bidimensionale) o in formato 1D (monodimensionale). Tuttavia, in C, gli array bidimensionali sono solo uno schema di indicizzazione accurato (é una rappresentazione intuitiva e facilmente comprensibile) per gli array unidimensionali. Proprio come con un array 1D, gli array 2D allocano un singolo blocco di memoria contigua e la notazione A[row][col] è simile a dire A[row*NCOLS+col].

Poiché la matrice X è memorizzata per righe ci ritroviamo la seguente struttura in memoria:

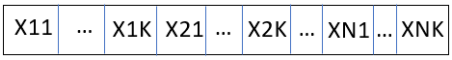


Figure 3: Rappresentazione in memoria lineare della matrice X

La funzione *transpose_from_2D()* calcola la trasposta della matrice X restituendo un array unidimensionale strutturato nel seguente modo:

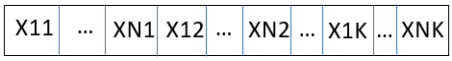


Figure 4: Rappresentazione in memoria lineare della matrice X trasposta

Come si può osservare, siamo riusciti ad ottenere la contiguità in memoria per le singole colonne della matrice X. Più precisamente, ritroviamo gli elementi della prima colonna, della seconda colonna e così via per le rimanenti colonne.
A questo punto, è stata calcolata la dimensione del *chunk* in modo da suddividere il numero di righe tra i vari threads. La dimensione del chunk (*chunk_size*) viene calcolata dalla funzione *compute_chunk_size* considerando il numero totale di righe M e il numero totale di thread disponibili (*nthreads*). L'idea è quella di assegnare ad ogni thread possibilmente lo stesso numero di righe della matrice sparsa (e quindi anche della matrice finale Y) e di assegnare ai threads un blocco di righe contiguo in modo da evitare il più possibile il problema del *False Cache Sharing* e da sfruttare il più possibile la località dei dati. Ad ogni thread viene associato un certo numero di righe e ogni riga è associata ad un unico thread. 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.
Poiché viene utilizzata la clausola *schedule* combinata con la modalità *static* e con la dimensione del chunk precedentemente calcolata, allora il primo thread si prende le prime *chunk_size* righe, il secondo thread si prende le seconde *chunk_size* righe e così via per i rimanenti thread qualora ci fossero.
Analizziamo che cosa viene fatto dal thread T a cui viene assegnata la riga *i-esima*. Come primo passo vengono calcolati i due valori *start* e *end* che rappresentano rispettivamente gli indici per gli array *as* e *ja* relativi al primo non zero della riga *i-esima* e della riga *(i+1)-esima*:

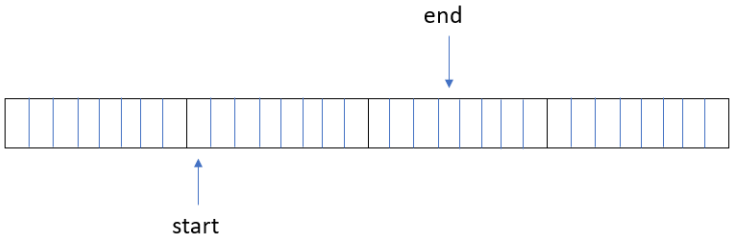


Figure 5: Non-zeri appartenenti alla riga *i-esima*

Osservo che i non zeri appartenenti ad una stessa riga (e.g., la riga *i-esima*) coprono posizioni contigue in memoria. Inoltre, osservo che ad uno stesso thread T gli possono essere assegnate più righe della matrice. Tuttavia, poiché le righe che vengono assegnate ai thread sono consecutive tra di loro (o almeno consecutive a blocchi) e i non zeri per ogni riga sono contigui in memoria allora il thread T lavorerà su una porzione contigua e sperabilmente grande (dipende dalla dimensione del chunk e dal numero di non zeri nelle righe assegnate al thread) delle strutture dati *as* e *ja*.
I thread accedono in lettura alle strutture dati *as*, *ja* e alla trasposta della matrice *X*. Poiché l'accesso a tali informazioni avviene in lettura e non in scrittura, tipicamente il protocollo di cache coherence sottostante consente di avere i blocchi nello stato *shared* per le varie cache L1. Nello stato "shared" (condiviso) di un protocollo di coerenza della cache, il blocco di memoria (cache line) è presente in diverse cache e le copie in queste cache condividono lo stesso valore. Questo stato si verifica quando un core esegue un'operazione di lettura dalla memoria e il blocco viene caricato nella sua cache, mentre altre cache che condividono lo stesso blocco continuano a mantenere copie valide. Di conseguenza, tutto ciò non mi dà problemi dal punto di vista del *False Cache Sharing*. Siccome in lettura nessun cache-controller effettua una RFO (request for ownership) di una linea di cache e quindi le altre copie (della stessa linea) non devono essere invalidate, allora non si ha il flooding di transazione distribuite sull'architettura di cache e di conseguenza non si ha degrado delle performance.
Inoltre, i vari thread possono sfruttare la località dei dati per via della loro contiguità in memoria.
Infine, passiamo a considerare la matrice Y. La matrice Y è memorizzata per righe. Nel codice vengono effettuate le scritture sulla matrice Y da parte dei vari threads. Poiché ai thread vengono assegnate le righe a blocchi contigui e gli elementi *Yij* sono memorizzati per righe allora riesco a ridurre anche il *False Cache Sharing* per la matrice Y.

3.3 CUDA

Per quanto riguarda il prodotto parallelo per il formato *CSR* in *CUDA*, abbiamo deciso di scrivere differenti versioni del kernel. Nella funzione *CSR_GPU()* vengono eseguite delle operazioni di setup per poter poi invocare il kernel di in-

teresse. A seconda della modalità di compilazione (e.g., l'algoritmo che si vuole utilizzare) vengono definite specifiche variabili che saranno successivamente utilizzare per il prodotto parallelo. Tra le operazioni di setup possiamo considerare:

- Allocazione della memoria lato GPU.
- Trasferimento dei dati dall'host alla GPU.
- Calcolo del numero di blocchi da utilizzare per il prodotto parallelo.

Ovviamente, a seguito della completa esecuzione del kernel il risultato viene copiato dalla GPU all'host. Inoltre, viene gestito lo spazio sulla GPU deallocando la memoria utilizzata per il calcolo precedente. Infine, la memoria viene deallocata anche lato host.

Le prime tre versioni del kernel potrebbero essere viste come la stessa versione del *CSR_Scalar* con solamente delle modifiche minimali che le distinguono. Si è 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 grande impatto sulle prestazioni.
L'idea che c'è dietro alle prime tre versioni del kernel è quella di far sì che un singolo thread T computi un singolo elemento Yiz della matrice finale Y. Il numero di threads per blocco che viene utilizzato è pari a *512*. Per stabilire il miglior numero di thread per blocco, abbiamo eseguito delle prove empiriche che hanno rivelato un miglioramento delle prestazioni 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 dovrà essere calcolato dal thread. Come primo passo viene calcolato l'identificativo globale del thread (tid). Successivamente, vengono calcolati i due indici dell'elemento 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 Y.
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 alla memoria 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* per poi 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 tra più thread la computazione richiesta per il calcolo di un singolo elemento della matrice *Y*. Il quarto kernel *CSR_Vector()* è ispirato all'algoritmo del *CSR Vector* descritto nei vari paper. L'idea è quella di distribuire il calcolo di uno stesso 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*. Poiché i thread appartenenti allo stesso warp si occupano di calcolare lo stesso elemento *(i,z)* della matrice *Y*, viene utilizzato l'identificativo globale del warp per determinare gli indici dell'elemento da calcolare. Poiché è possibile che siano stati generati più warp degli elementi della matrice *Y* allora viene fatto un controllo sul valore dell'identificativo del warp. Viene eseguita una riduzione parallela che coinvolge i thread di uno stesso warp sfruttando la *shared memory*. Ogni thread scrive il proprio risultato parziale nella locazione di memoria condivisa che gli è stata assegnata. La zona di memoria condivisa viene assegnata ai thread in base all'identificativo locale del thread all'interno del blocco a cui appartiene. Siccome tutti i threads di un warp contribuiscono al calcolo dello stesso elemento, è sufficiente che un solo thread tra essi scriva in memoria globale il risultato ottenuto dalla riduzione. Per fare ciò, si è 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 rispetto a quelle che abbiamo ottenuto con il *CSR_Scalar()*. Il motivo per cui le prestazioni si sono ridotte così drasticamente è dovuto principalmente al basso numero 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 per riga che è inferiore rispetto alla dimensione del warp. Questo implica che ci siano mediamente un numero non necessariamente basso 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. Infatti, rispetto alle tre versione del kernel che sono state descritte precedentemente, abbiamo lo stesso numero complessivo di accessi in memoria globale ma, in più, un notevole numero di accessi in memoria condivisa.

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 intero 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 *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 che compongono uno stesso sub-warp. La dimensione del subwarp è un valore scelto dopo un'attenta analisi empirica delle prestazioni al variare del numero di thread per ogni sub-warp. Infatti, nell'analisi è stato osservato come al crescere della dimensione del sub-warp le prestazioni tendono a diminuire. Nella versione del kernel *CSR_Vector_Sub_warp()* si ragiona in termini di identificativo del sub-warp. Come primo passo si determina l'identificativo globale del thread che verrà poi utilizzato per 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 valori per l'identificativo dei threads all'interno di un sub-warp sono *0* e *1*. Dopo aver calcolato l'indice globale del sub-warp è possibile determinare la riga e la colonna dell'elemento della matrice che verrà assegnato al sub-warp. Poiché i thread che compongono il sub-warp si occupano di calcolare le somme parziali per un singolo elemento della matrice Y allora possiamo associare ad ogni thread una singola area nella memoria condivisa per memorizzare i risultati parziali. Da notare che questo non sarebbe stato possibile se i thread del sub-warp avessero calcolato le somme parziali relative a 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. Dopo aver calcolato i vari 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.

Infine, è stata implementata un'ultima variante del *CSR Vector* rappresentata da *CSR_Vector_by_row*. Per quanto riguarda i due kernel *CSR_Vector* e *CSR_Vector_Sub_warp*, il calcolo di un singolo elemento viene distribuito tra i thread di un intero warp oppure tra i thread di un sub-warp. Nel caso di questa terza versione del *CSR Vector* si è deciso di provare un altro modo per suddividere il carico di lavoro: ogni warp è responsabile di calcolare tutte le componenti di un singola riga della matrice Y. Come si può osservare, l'idea non è quella di parallelizzare il calcolo di un singolo elemento della matrice Y ma quella di parallelizzare il calcolo dei K elementi che fanno parte di una stessa riga della matrice Y. Si è deciso di implementare questa versione anche per il fatto che il numero di elementi K che compongono una riga è molto limitato. A questo punto, scendiamo un pò più nel dettaglio dell'algoritmo. Ogni warp, suddiviso in 8 sub-warp, è responsabile del calcolo di tutti le componenti di una riga della matrice Y. I thread del sub-warp collaborano per calcolare uno stesso

elemento la cui riga è determinata dall'identificativo del warp di cui il sub-warp fa parte. Di conseguenza, il calcolo degli elementi della matrice Y è distribuito tra i vari sub-warp. Viene eseguita una riduzione in parallelo e solamente il thread con identificativo pari a *0* nel sub-warp ha il compito di scrivere il risultato nella memoria globale.
In breve, per ogni warp abbiamo 8 sub-warp composti ciascuno da 4 thread che si suddividono le K colonne di X per calcolare una intera riga della matrice *Y*.

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 la 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-Adaptive 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, un numero di threads pari al numero di righe per il blocco corrente eseguono la riduzione scalare dalla memoria condivisa per calcolare l'output finale.

Anche in questo caso, non soddisfatti delle prestazioni ottenute abbiamo realizzato la seguente versione ottimizzata per spMM di CSR-Adaptive:
CSR_Adaptive_sub_blocks. Ogni blocco di thread, suddiviso in un certo numero

di sub-block, calcola tutte le componenti di una riga della matrice Y. Ogni sub-block di sub_block_size thread è responsabile del calcolo di una prodotto della componente di sua competenza per la riga i nella matrice *Y*. Di conseguenza, i thread dello stesso sub-block collaborano tra di loro per calcolare l'output finale. Negli algoritmi CSR_Vector_by_row, CSR Vector e CSR Adaptive la matrice X è stata trasposta in modo tale che gli elementi della colonna siano contigui in memoria.

In questo modo abbiamo ridotto il numero di conflitti sulla gerarchia di cache per l'accesso alle colonne della matrice X. Di fatto, abbiamo meno threads che possono accedere in concorrenza alla stessa colonna della matrice X.

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 passo viene calcolato il massimo numero degli elementi non-zero (*max_so_far*) tra tutte le righe della matrice sparsa. 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 a 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 per quanto riguarda la 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 calcolato il numero degli elementi non-zero all'interno di ciascuna riga della matrice popolando il vettore *nz_per_row*. Successivamente, abbiamo allocato la memoria per le due strutture dati *values* e *col_indices* sulla base del numero di non zeri calcolato precedentemente senza inserire alcun padding.

Sebbene la seconda versione ci ha permesso di risparmiare memoria risolvendo il problema dell'esecuzione locale, entrambe le versioni precedenti richiedono molto tempo per la conversione dal formato *COO* al formato *ELLPACK*, soprattutto considerando le matrici di grandi dimensioni. Per superare quest'ultimo ostacolo, abbiamo deciso di implementare una terza versione dell'algoritmo di conversione che ci permette 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 passo consiste sempre nel computare 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 lavorare su una stessa riga.

4.2 OpenMP

Il prodotto parallelo per il formato *ELLPACK* con parallelizzazione *OpenMP* è stato implementato in due varianti distinte 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. Nel funzione *main* viene invocata la funzione *create_dense_matrix* che crea una matrice densa di valori casuali. La matrice X ha N righe e K colonne

Figure 6: Matrice Densa X

ed è memorizza per righe

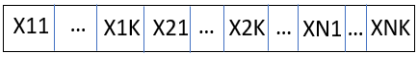


Figure 7: Matrice Densa X nella memoria lineare

Con la funzione *transpose_from_2D()* viene calcolata la trasposta della matrice X restituendo un array unidimensionale strutturato nel seguente modo:

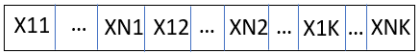


Figure 8: Matrice trasposta di X nella memoria lineare

Viene calcolata la dimensione del chunk (*chunk_size*) tramite la funzione *compute_chunk_size()* in modo da suddividere le M righe della matrice sparsa A tra i vari threads. La funzione suddetta prende in input la *la dimensione dello spazio d'iterazione* e il numero di thread disponibili (*nthread*) e restituisce:

- *(iter_space_dim / nthread)* se M % nthread = 0
- *(iter_space_dim / nthread + 1)* se M % nthread != 0.

La dimensione del chunk è stata calcolata in questo modo con lo scopo di raggiungere i seguenti due obiettivi:

- Assegnare un buon numero di righe consecutive ad ogni thread in modo da sfruttare meglio la località dei dati.
- Bilanciare le righe tra i thread disponibili nel modo più uniforme possibile.

Per studiare il comportamento dell'algoritmo relativamente alla località dei dati e al False Cache Sharing riprendiamo le strutture dati e le informazioni che verranno utilizzate:

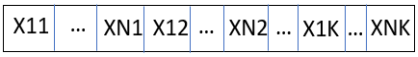
- *max_nz_per_row*: è il numero massimo di non zeri nelle righe della matrice sparsa.
- *ja*: è un array 2D di indici di colonna che ha M righe e *max_nz_per_row* colonne con l'aggiunta di un opportuno padding per le righe della matrice A che hanno un numero di non zeri inferiore a *max_nz_per_row*.
- *as*: è un array 2D contenente i coefficienti non zero della matrice A con l'aggiunta di uno zero padding per le righe della matrice A che hanno un numero di non zeri inferiore a *max_nz_per_row*.
- La matrice Y
- La trasposta della matrice X

Le strutture dati *ja*, *as* e la trasposta di X sono accedute solamente in lettura e sono condivise tra i vari threads. Tipicamente avere solo accessi in lettura non è problematico dal punto di vista prestazionale per quanto riguarda il protocollo di cache coherency sottostante poiché è possibile avere un blocco di dati valido in più cache L1. Viene utilizzata la clausola *schedule* insieme alla dimensione del chunk precedentemente calcolata e alla modalità *static*. Di conseguenza, i thread accedono a zone di memoria contigue. Per capire meglio come tutto ciò accade, scendiamo un pò più nei dettagli con le singole strutture dati.

Le strutture dati *as* e *ja* sono degli array bidimensionali memorizzati per righe nella memoria lineare. Per quanto riguarda la matrice X abbiamo:

$$X = \begin{bmatrix} X11 & X12 & \dots & X1K \\ X21 & X22 & \dots & X2K \\ & & \dots & \\ XN1 & XN2 & \dots & XNK \end{bmatrix}$$

$$X^T = \begin{bmatrix} X11 & X21 & \dots & XN1 \\ X12 & X22 & \dots & XN2 \\ & & \dots & \\ X1K & X2K & \dots & XNK \end{bmatrix}$$



Per come è stata calcolata la dimensione del chunk e considerando l'utilizzo della modalità *static* abbiamo che blocchi di righe consecutive vengono assegnati ai thread disponibili. Ad esempio, se $M = 999$ e $nthread = 5$ allora abbiamo:

- $chunk\_size = 999/5 + 1 = 200$
- Le righe vengono distribuite nel seguente modo:
  - T_0: [1, 200]
  - T_1: [201, 400]
  - T_2: [401, 600]
  - T_3: [601, 800]
  - T_4: [801, 999]

Poiché gli elementi appartenenti alla stessa riga della matrice A sono contigui in memoria e le righe sono memorizzate consecutivamente (per indice) nelle strutture dati *ja* e *as* allora gli elementi di righe consecutive sono contigui in memoria. Riprendendo l'esempio precedente, al thread T_0 vengono assegnate le righe dalla 1 alla 200. Di conseguenza, esso accederà agli elementi

- $as[0][j], as[1][j], \ldots, as[199][j]\ \forall j = 0, ..., max\_nz\_per\_row$
- $aj[0][j], aj[1][j], ..., aj[199][j]\ \forall j = 0, ..., max\_nz\_per\_row$

che sono contigui in memoria. In questo modo, siamo in grado di sfruttare la località dei dati nella cache.

A questo punto, consideriamo la trasposta della matrice X. Per calcolare l'elemento Y[i][j] viene eseguito il seguente calcolo:

$$Y[i][j] = a_{i1} * x_{1j} + ... + a_{iN} * x_{Nj}$$

Poiché lavoriamo con la trasposta della matrice X allora gli elementi della *j-esima* colonna sono contigui in memoria e riesco a sfruttare la località dei dati.

Per quanto riguarda il *False Cache Sharing* dobbiamo porre attenzione sulle scritture che vengono effettuate per la matrice Y. La matrice Y è memorizzata per righe nella memoria lineare. Lo scenario migliore che mi consente di ridurre la probabilità del *False Cache Sharing* lo abbiamo nel momento in cui un thread scrive elementi della matrice Y che appartengono alla stessa riga e che quindi sono contigui in memoria. Considerando il nostro algoritmo, osserviamo che se al thread T viene assegnata la riga *i-esima* allora esso ha la responsabilità di determinare le componenti Y[i][z] con *z = 0, ..., K - 1*. Più precisamente, il

thread dovrà occuparsi di tutti gli elementi della riga *i-esima*. Poiché ai thread viene assegnato un blocco di righe consecutive allora riusciamo a ridurre la probabilità del *False Cache Sharing*.

La differenza tra le due funzioni

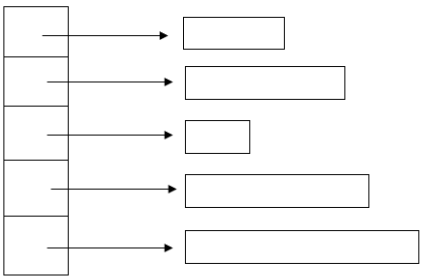
- *parallel_product_ellpack()*
- *parallel_product_ellpack_no_zero_padding()*

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

Come descritto nelle sezioni precedenti, sono state realizzate due differenti versioni per il formato *ELLPACK*:

- Una versione che utilizzare il padding per le strutture dati *ja* e *as*. In questo caso, le due strutture dati rappresentano degli array bidimensionali che hanno un numero di righe pari al numero di righe M della matrice sparsa A e un numero di colonne pari al numero massimo di non zeri per riga della matrice sparsa A.
- Una versione che non utilizza il padding. In questo caso, le due strutture dati *ja* e *as* non necessariamente possono essere viste come matrici. Infatti, è possibile che ci siano delle righe con un differente numero di non zeri.



Si è deciso di utilizzare la versione ottimizzata del formato *ELLPACK* che non utilizza il padding in modo da gestire meglio la memoria presente sulla GPU. Tutto ciò vale soprattutto per le matrici di grandi dimensioni. Per convertire queste strutture dati in un array *1D* viene utilizzata la funzione *convert_2D_to_1D_per_ragged_matrix()* che utilizza come informazione il numero di

non zeri per riga.

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 è 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 sfruttando l'identificativo globale del thread stesso (*tid*):

  $$row = tid \ / \ K$$

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

  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 è strettamente minore del numero di elementi della matrice *Y* poiché è possibile che siano stati generati più thread di quelli effettivamente necessari.
  Supponiamo che al thread sia stato assegnato il compito di computare un elemento della matrice *Y* che ha il valore *i* come indice di riga. Il thread deve recuperare la posizione nelle strutture dati *d_values (as)* e *d_col_indices (ja)* relativa al primo elemento non zero della riga *i-esima*. Per fare ciò utilizza la struttura dati *sum_nz*. La entry *i-esima* della struttura dati sum_nz rappresenta il numero di non zeri che si hanno fino alla riga *i-esima* esclusa. Di conseguenza, questo valore può essere utilizzato come *offset* all'interno delle due strutture dati. A questo punto, la struttura dati *nz_per_row* viene utilizzata per recuperare il numero di non zeri contenuti all'interno della riga *i-esima*. Questo valore consente di definire l'estremo finale del ciclo.

- *ELLPACK_Sub_warp()*: Vengono utilizzati *16* differenti *sub-warp* composti da 2 thread ciascuno. Ogni sub-warp ha il compito di calcolare un singolo elemento della matrice finale *Y*. Di conseguenza, ogni thread all'interno di un sub-warp calcola un risultato parziale per uno specifico elemento della matrice *Y*. Come primo passo 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 rispetto al sub-warp che può assumere come valore *0* oppure *1*. Dopo aver calcolato l'identificativo globale del sub-warp è possibile determinare la riga e la colonna dell'elemento della matrice *Y* che è stato assegnato al sub-warp. Poiché i thread che compongono un sub-warp si occupano

di computare le somme parziali per un singolo elemento della matrice Y allora possiamo associare ad ogni thread nel sub-warp una singola area nella memoria condivisa (come descritto precedentemente). 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 identificativo *0* all'interno del sub-warp scriverà in memoria globale il risultato finale.

5 Misurazione delle prestazioni

In questo capitolo, ci concentreremo sull'analisi prestazionale del prodotto *matrice-multivettore* su CPU e GPU. Per comprendere meglio il comportamento degli algoritmi e per studiarne le prestazioni abbiamo realizzato dei grafici appositi. Come da traccia, le prestazioni sono state valutate sul server del dipartimento.

La banda di confidenza del *95%* è stata inserita nell'analisi delle prestazioni del prodotto matrice sparsa con multivettore su CPU e GPU in modo da avere un'indicazione sulla variabilità dei risultati delle misurazioni. Quando si eseguono misurazioni sperimentali, è comune osservare una certa variabilità nei risultati a causa di diversi fattori, come le condizioni di carico del sistema, l'architettura hardware specifica, la gestione delle risorse da parte del sistema operativo, ecc. La banda di confidenza rappresenta l'intervallo di valori entro il quale ci si può aspettare che ricadano i risultati delle misurazioni con un determinato livello di confidenza statistica. La sua inclusione nel contesto dell'analisi delle prestazioni aiuta a comprendere meglio l'incertezza associata ai risultati e a fornire una stima della robustezza delle misurazioni.

5.1 Prestazioni su CPU

Data la matrice in input, i grafici realizzati su CPU mostrano l'andamento del tempo medio di esecuzione espresso in secondi al variare del numero di thread e del numero di colonne K della matrice X. Il codice è stato collaudato con un numero di thread variabile da 1 fino al massimo numero di core disponibili sulla piattaforma di calcolo. Inoltre, il codice è stato progettato e implementato per funzionare con un k generico ed è stato collaudato per i valori di k nella lista [3, 4, 8, 12, 16, 32, 64]. Ci aspettiamo un andamento crescente del tempo medio di esecuzione al crescere del numero di colonne K e un andamento decrescente del tempo medio di esecuzione al crescere del numero di thread utilizzati per il calcolo.

5.1.1 Analisi prestazionale per il formato CSR

Il comportamento atteso descritto nella sezione precedente è evidente nella seguente immagine:

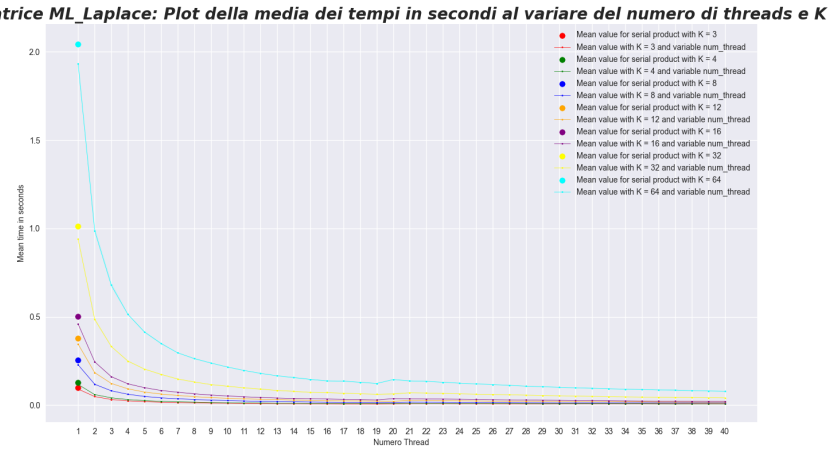


Figure 9: Tempi medi per la matrice *ML_Laplace* nel formato CSR

Il grafico precedente mostra l'andamento del tempo medio di esecuzione al variare del numero di thread e del numero di colonne K per la matrice *ML_Laplace*. I singoli punti colorati mostrano il tempo di esecuzione medio per il prodotto seriale al variare di K. Ogni colore è associato ad uno specifico valore del numero di colonne K. Come si può osservare dal grafico:

1 fissato il numero di colonne K dela matrice X, il tempo medio di esecuzione decresce all'aumentare del numero di thread.

2 fissato il numero di thread, il tempo di esecuzione medio aumenta all'aumentare del numero di colonne K della matrice X.

La spiegazione di questi due comportamenti è molto intuitiva:

1 L'utilizzo di un numero maggiore di thread consente di sfruttare meglio il parallelismo disponibile nel sistema. Inoltre, consente una migliore distribuzione del carico di lavoro tra le varie unità di elaborazione. Con l'aumentare del numero di thread disponibili le attività possono essere eseguite contemporaneamente su più unità di elaborazione riducendo il tempo complessivo di esecuzione. Tuttavia, è importante notare che tipicamente l'aumento del numero di thread potrebbe comportare anche un overhead aggiuntivo dovuto alla sincronizzazione tra i thread e alla gestione degli accessi da parte dei thread alle risorse condivise.

2 Fissato il numero di thread, l'aumento del tempo medio di esecuzione al crescere del numero di colonne K del multivettore può essere attribuito a diversi fattori tra cui:

   – **Accesso alla memoria:** *L'elaborazione di un multi-vettore richiede l'accesso a K elementi del vettore per ogni colonna della matrice.* All'aumentare del numero di colonne K, aumenta anche la quantità di dati che dovranno essere letti dalla memoria. Tutto ciò può comportare un maggior numero di *cache miss* per cui i dati richiesti, non essendo presenti nella cache, dovranno essere recuperati dalla memoria principale. L'accesso alla memoria principale è servito molto più lentamente rispetto all'accesso alla memoria cache. L'aumento del numero di accessi in memoria influisce negativamente sul tempo di esecuzione complessivo.

   – **Complessità computazionale:** L'aumento del numero di colonne K del multivettore comporta un aumento della dimensione della matrice e, quindi, un aumento della quantità complessiva dei dati coinvolti nel prodotto tra la matrice e il multivettore. Di conseguenza, aumenta il numero di operazioni che dovranno essere svolte per risolvere il problema.

Un andamento simile si ha per quanto riguarda la matrice *adder_dcop_32* che, rispetto alla matrice *ML_Laplace*, ha un numero medio di non zeri per riga notevolmente inferiore:

- $\frac{11.246}{1.813} \approx 6,202$ per la matrice *adder_dcop_32*
- $\frac{27.689.972}{377.002} \approx 73,447$ per la matrice *ML_Laplace*

Il grafico seguente mostra l'andamento del tempo medio di esecuzione al variare del numero di thread e del numero di colonne K per la matrice *adder_dcop_32*.

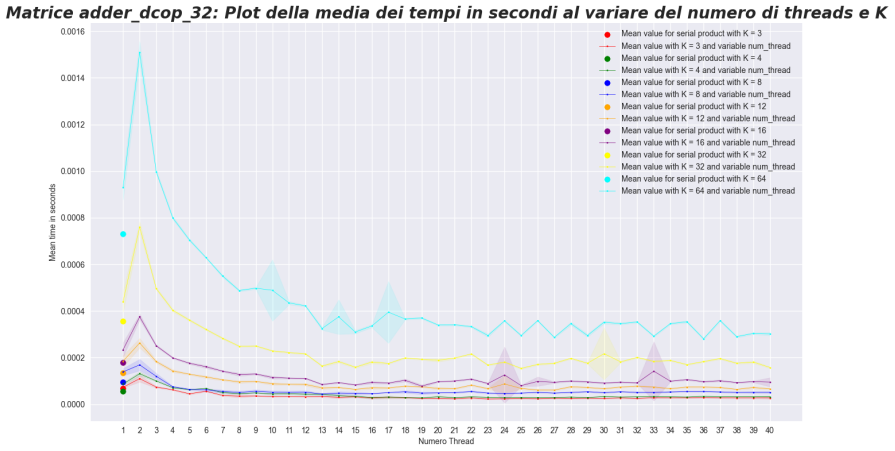


Figure 10: Tempi medi per la matrice *adder_dcop_32* nel formato CSR

5.1.2 Analisi prestazionale per il formato ELLPACK

Per quanto riguarda l'analisi prestazionale relativa al formato *ELLPACK* possiamo notare un andamento analogo per il tempo medio di esecuzione al variare del numero di thread e del numero di colonne K della matrice X. La spiegazione di questo andamento è la stessa descritta nella sezione precedente relativa al formato CSR. Di seguito sono stati riportati dei grafici rappresentanti l'andamento del tempo medio di esecuzione per alcune delle matrici di test.

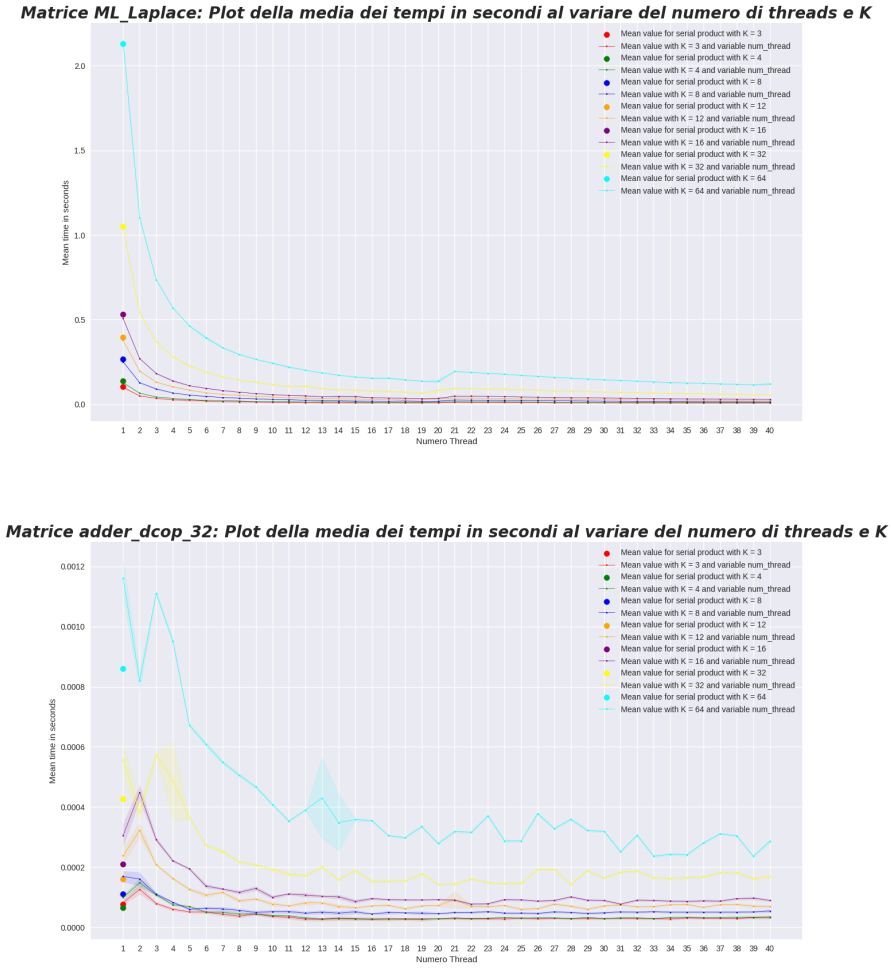


Figure 11: Tempi medi per la matrice ML_Laplace e adder_dcop_32 nel formato ELLPACK

5.2 Prestazioni su GPU

La complessità intrinseca delle GPU ha reso difficile individuare un "pattern" ricorrente nelle trame dei prodotti matrice sparsa con multi-vettore. La GPU è un'architettura altamente specializzata progettata per eseguire in modo efficiente operazioni parallele su grandi quantità di dati. A differenza delle CPU, le GPU sono caratterizzate da un elevato grado di parallelismo massivo (quindi maggiore variabilità) e da una gerarchia di memoria complessa.

A causa di questa complessità, l'analisi delle prestazioni sulle GPU richiede un approccio dettagliato e specifico per l'architettura utilizzata. Le GPU moderne possono avere diverse generazioni, modelli e configurazioni, ognuna con le proprie caratteristiche uniche. Pertanto, è difficile identificare un pattern generale nei plots dei prodotti matrice sparsa con multi-vettore su GPU che sia applicabile a tutte le architetture.

Per quanto riguarda la GPU, abbiamo realizzato dei grafici che mostrano l'andamento medio dei GFLOPS al variare del kernel utilizzato e di K. Anche in questo caso abbiamo rappresentato la banda di confidenza del 95%.

5.2.1 GPU e CSR

Consideriamo per primo il formato CSR. Come detto in precedenza, per questo formato abbiamo realizzato diversi kernel. Per analizzarli conviene confrontare i risultati ottenuti per le matrici adder_dcop_32 e ML_Laplace, essendo dimensionalmente e strutturalmente molto diverse.

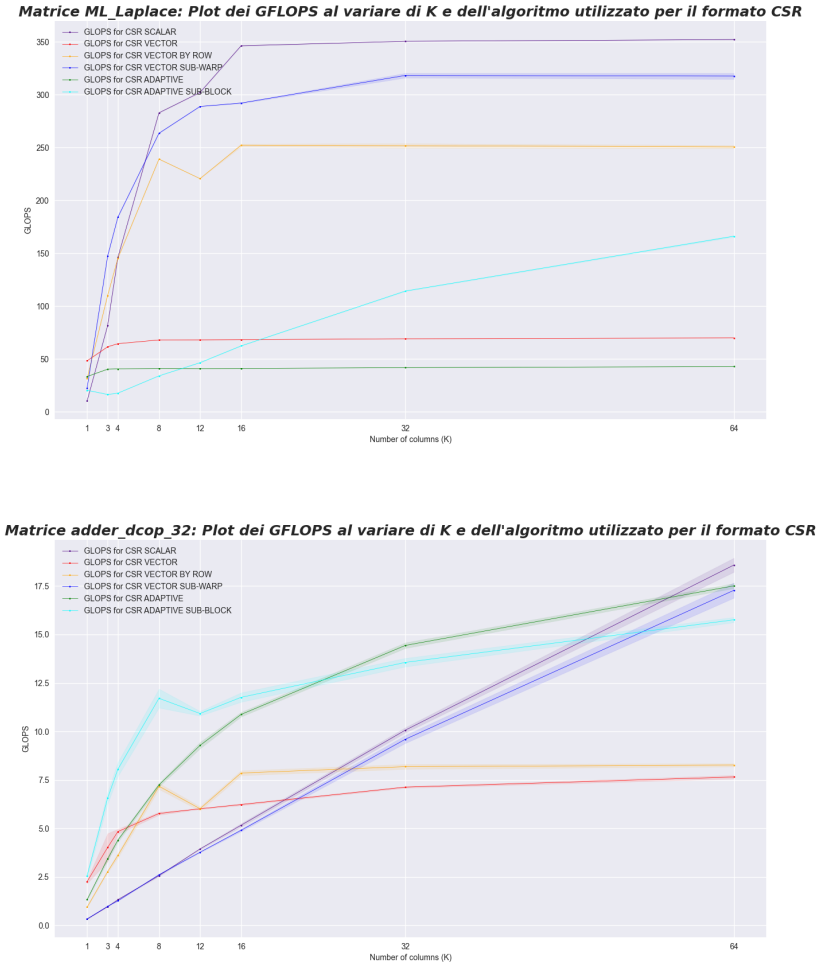


Figure 12: GFLOPS per la matrice ML_Laplace e adder_dcop_32 nel formato CSR

Un fattore determinante per analizzare le immagini precedenti è il numero medio di non zeri:

- Per ML_Laplace è $\frac{27.689.972}{377.002} \approx 73,447$
- Per adder_dcop_32 $\frac{11.246}{1.813} \approx 6,202$

E' possibile fare le seguenti considerazioni algoritmo per algoritmo: (TODO)

- **CSR_SCALAR**

- **CSR_VECTOR**
- **CSR_VECTOR_BY_ROW**
- **CSR_VECTOR_SUB_WARP**
- **CSR_ADAPTIVE**
- **CSR_ADAPTIVE_SUB_BLOCK**

5.2.2 GPU e ELLPACK

Per quanto riguarda il formato ELLPACK abbiamo la seguente situazione:

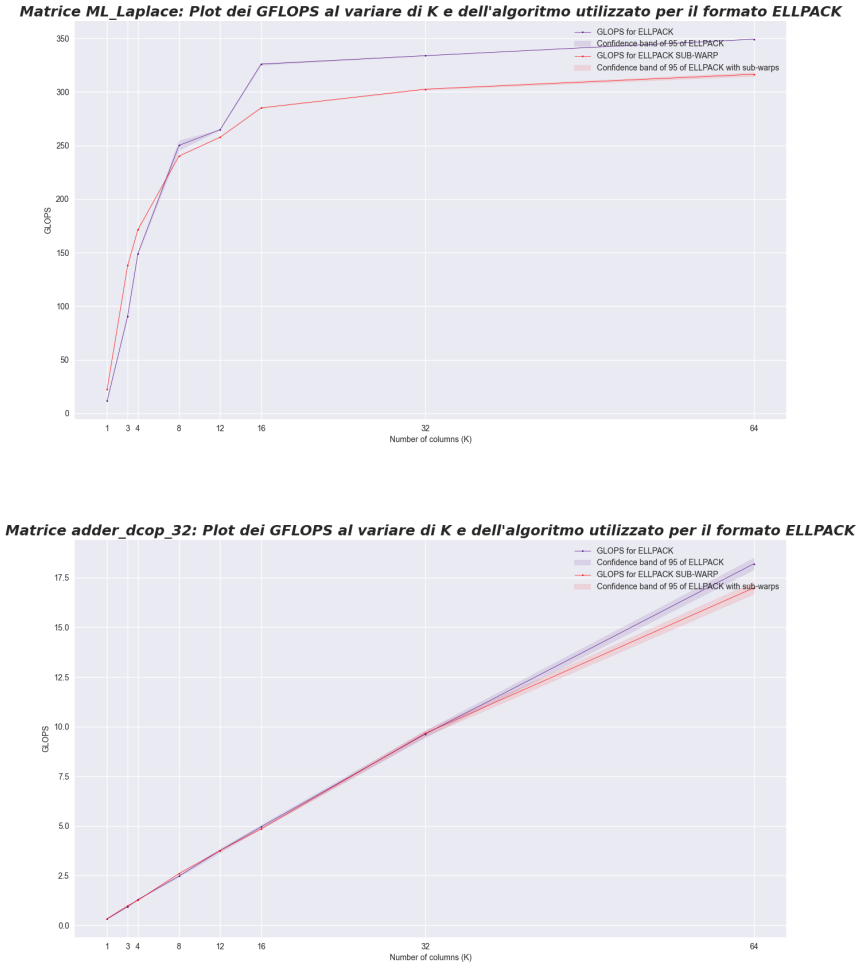


Figure 13: GFLOPS per la matrice ML_Laplace e adder_dcop_32 nel formato ELLPACK

E' possibile fare le seguenti considerazioni algoritmo per algoritmo: (TODO)

- **ELLPACK_SCALAR**
- **ELLPACK_SUB_WARP**

6 Suddivisione del lavoro

Il progetto è stato congiuntamente elaborato durante l'ultima sessione didattica. Di conseguenza, ci siamo incontrati giornalmente, dovendo seguire gli stessi corsi, per portare avanti il progetto e risolvere eventuali problematiche incontrate. Di fatto, non c'è una netta suddivisione del carico di lavoro.

7 Istruzioni per la compilazione

Per quanto riguarda la compilazione abbiamo reso disponibile nella root directory del progetto un Makefile. Il suo utilizzo può essere esemplificato nel seguente modo:

- Compilazione openMP, a sua volta schematizzata nel seguente modo:
  - **openmp-csr-compare-serial-parallel**:

```
1       #!/bin/bash
2       make openmp-csr-compare-serial-parallel
```

    Con il precedente comando il codice viene compilato per eseguire sia il prodotto seriale che parallelo per il formato CSR, confrontare i tempi d'esecuzione e verificare la correttezza di quello parallelo attraverso un confronto elemento per elemento con quello seriale.

  - **openmp-ellpack-compare-serial-parallel**

```
1       #!/bin/bash
2       make openmp-ellpack-compare-serial-parallel
```

    Con il precedente comando il codice viene compilato per ottenere un eseguibile che calcoli sia il prodotto seriale che parallelo per il formato ELLPACK, confrontare i tempi d'esecuzione e verificare la correttezza di quello parallelo attraverso un confronto elemento per elemento con quello seriale.

  - **openmp-csr-check-conversions**

```
1       #!/bin/bash
2       make openmp-csr-check-conversions
```

    Con il precedente comando il codice viene compilato per confrontare la conversione originale e quella ottimizzata dal formato COO al formato CSR.

  - **openmp-ellpack-check-conversions**

```
1       #!/bin/bash
2       make openmp-ellpack-check-conversions
```

Con il precedente comando il codice viene compilato per confrontare la conversione originale e quella ottimizzata dal formato COO al formato ELLPACK.

- **openmp-csr-serial-samplings**

```
#!/bin/bash
make openmp-csr-serial-samplings
```

Con il precedente comando il codice viene compilato per ottenere un eseguibile che esegua al variare di K e del numero di thread, per SAMPLING_SIZE volte il prodotto seriale, calcoli la media e la varianza dei tempi d'esecuzione in modo incrementale ed in un singolo passo attraverso l'algoritmo di (one-pass) Welford

- **openmp-ellpack-serial-samplings**

```
#!/bin/bash
make openmp-ellpack-serial-samplings
```

Con il precedente comando il codice viene compilato per ottenere un eseguibile che esegua al variare di K e del numero di thread, per SAMPLING_SIZE volte il prodotto seriale per il formato ELLPACK, calcoli la media e la varianza dei tempi d'esecuzione in modo incrementale ed in un singolo passo attraverso l'algoritmo di (one-pass) Welford

- **openmp-csr-parallel-samplings**

```
#!/bin/bash
make openmp-csr-parallel-samplings
```

Con il precedente comando il codice viene compilato per ottenere un eseguibile che esegua al variare di K e del numero di thread, per SAMPLING_SIZE volte il prodotto parallelo per il formato CSR, calcoli la media e la varianza dei tempi d'esecuzione in modo incrementale ed in un singolo passo attraverso l'algoritmo di (one-pass) Welford

- **openmp-ellpack-parallel-samplings**

```
#!/bin/bash
make openmp-ellpack-parallel-samplings
```

Con il precedente comando il codice viene compilato per ottenere un eseguibile che esegua al variare di K e del numero di thread, per SAMPLING_SIZE volte il prodotto parallelo per il formato ELLPACK, calcoli la media e la varianza dei tempi d'esecuzione in modo incrementale ed in un singolo passo attraverso l'algoritmo di (one-pass) Welford

- Compilazione CUDA, a sua volta schematizzata nel seguente modo:

– **csr**

```
#!/bin/bash
make all MODE=csr
```

Con il precedente comando si ottiene un eseguibile che lancia il prodotto seriale, il kernel CUDA *CSR_Scalar_v3*, calcoli i GFLOPS e confronti il risultato del prodotto seriale con quello parallelo.

– **csr_adaptive**

```
#!/bin/bash
make all MODE=csr_adaptive
```

Con il precedente comando si ottiene un eseguibile che lancia il prodotto seriale, il kernel CUDA *CSR_Adaptive*, calcoli i GFLOPS e confronti il risultato del prodotto seriale con quello parallelo.

– **csr_adaptive_sub_block**

```
#!/bin/bash
make all MODE=csr_adaptive_sub_block
```

Con il precedente comando si ottiene un eseguibile che lancia il prodotto seriale, il kernel CUDA *CSR_Adaptive_sub_blocks*, calcoli i GFLOPS e confronti il risultato del prodotto seriale con quello parallelo.

– **csr_vector**

```
#!/bin/bash
make all MODE=csr_vector
```

Con il precedente comando si ottiene un eseguibile che lancia il prodotto seriale, il kernel CUDA *CSR_Vector*, calcoli i GFLOPS e confronti il risultato del prodotto seriale con quello parallelo.

– **csr_vector_by_row**

```
#!/bin/bash
make all MODE=csr_vector_by_row
```

Con il precedente comando si ottiene un eseguibile che lancia il prodotto seriale, il kernel CUDA *CSR_Vector_by_row*, calcoli i GFLOPS e confronti il risultato del prodotto seriale con quello parallelo.

– **csr_vector_sub_warp**

```
#!/bin/bash
make all MODE=csr_vector_sub_warp
```

Con il precedente comando si ottiene un eseguibile che lancia il prodotto seriale, il kernel CUDA *CSR_Vector_Sub_warp*, calcoli i GFLOPS e confronti il risultato del prodotto seriale con quello parallelo.

– **ellpack_sw**

```
#!/bin/bash
make all MODE=ellpack_sw
```

Con il precedente comando si ottiene un eseguibile che lancia il prodotto seriale, il kernel CUDA *ELLPACK_Sub_warp*, calcoli i GFLOPS e confronti il risultato del prodotto seriale con quello parallelo.

– **ellpack**

```
#!/bin/bash
make all MODE=ellpack
```

Con il precedente comando si ottiene un eseguibile che lancia il prodotto seriale, il kernel CUDA *ELLPACK_kernel*, calcoli i GFLOPS e confronti il risultato del prodotto seriale con quello parallelo.

✓ **Senza campionamento**

```
#!/bin/bash
make all SAMPLING=no
```

Con il precedente comando si ottiene un eseguibile che lancia il prodotto seriale, il kernel CUDA indicato nella variabile MODE (riga 19 nel Makefile), calcoli i GFLOPS e confronti il risultato del prodotto seriale con quello parallelo.

✓ **Con campionamento**

```
#!/bin/bash
make all SAMPLING=yes
```

Con il precedente comando si ottiene un eseguibile che lancia al variare di K, per SAMPLING_SIZE volte, il kernel CUDA indicato nella variabile MODE (riga 19 nel Makefile), calcoli la media e la varianza GFLOPS in modo incrementale ed in un singolo passo attraverso l'algoritmo di (one-pass) Welford

Una volta generato l'eseguibile *app* nella directory /all è possibile utilizzare nuovamente il comando make nel seguente modo:

```
#!/bin/bash
make ML_Laplace
```

.

In questo caso viene avviato il programma con in input la matrice ML_Laplace.

8 Struttura del codice e sua riutilizzabilità

Il codice è organizzata nelle seguenti directory:

- *CUDA*: In questa directory sono presenti 3 files :
  - *parallel_product_CSR.cu*: Contiene la funzione di setup e il codice sorgente dei kernel per il formato CSR.
  - *parallel_product_ELLPACK.cu*: Contiene la funzione di setup e il codice sorgente dei kernel per il formato ELLPACK.
  - *samplings.cu*: Contiene le funzioni implementare per effettuare il sampling per il nucleo di calcolo realizzato in CUDA.
- *doc*: E' la directory che contiene la relazione e le immagini utilizzate per realizzarla.
- *include*: E' la directory che contiene gli headers.
- *lib*: E' la directory che contiene la libreria ausiliaria *mmio.c*
- *openMP*: In questa directory sono presenti 3 files :
  - *parallel_product.c*: Contiene i codici che implementano i prodotti paralleli in openMP per il formato ELLPACK e CSR.
  - *samplings.c*: Contiene le funzioni implementare per effettuare il sampling per il nucleo di calcolo realizzato in openMP.
- *plots*: E' la directory che contiene i file python realizzati per sviluppo dei grafici, i plot nel formato .png e i file CSV con il risultato del campionamento.

Nella directory root, quella più "esterna" abbiamo i seguenti files:

- *checks.c*: Contiene le funzioni necessarie per effettuare i controlli sulla correttezza delle conversioni e del prodotto parallelo.
- *conversions_parallel.c*: E' presente il codice sviluppato per realizzare le conversioni in modo parallelo dal formato COO al formato CSR ed ELLPACK.
- *conversions_serial.c* E' presente il codice sviluppato per realizzare le conversioni in modo seriale dal formato COO al formato CSR ed ELLPACK.
- *create_mtx_coo.c*: Sono presenti le funzioni che implementano la lettura del file .mtx e conseguente allocazione di memoria per memorizzare la matrice nel formato COO.
- *main.c*: E' il file che contiene il main, che si comporta come orchestratore del programma.

- *Makefile*: E' il Makefile descritto nella sezione precedente.
- *serial_product.c*: Contiene le funzioni implementate per realizzare il prodotto seriale per il formato CSR ed ELLPACK.
- *utils.c*: Contiene le funzioni di utilità implementate.