

Prodotto tra una matrice sparsa ed un multivettore

Luca Capotombolo, Ludovico Zarrelli

1 Luglio 2023

Contents

| | | |
|----------|---|-----------|
| 1 | Introduzione | 3 |
| 1.1 | Presentazione del problema | 3 |
| 1.2 | Matrici di test e funzioni di I/O | 3 |
| 1.3 | Implementazione Seriale | 3 |
| 2 | Funzioni ausiliarie per il pre-processamento dei dati | 3 |
| 3 | Discussione implementazione CSR | 4 |
| 3.1 | Funzioni ausiliarie per la memorizzazione nel formato CSR | 4 |
| 3.2 | OpenMP | 5 |
| 3.3 | CUDA | 8 |
| 4 | Discussione implementazione ELLPACK | 13 |
| 4.1 | Funzioni ausiliarie per la memorizzazione nel formato ELLPACK | 14 |
| 4.2 | OpenMP | 15 |
| 4.3 | CUDA | 18 |
| 5 | Misurazione delle prestazioni | 20 |
| 5.1 | Prestazioni su CPU | 20 |
| 5.1.1 | Analisi prestazionale per il formato CSR | 20 |
| 5.1.2 | Analisi prestazionale per il formato ELLPACK | 27 |
| 5.2 | Prestazioni su GPU | 33 |
| 5.2.1 | Campionamento | 33 |
| 5.2.2 | CSR | 34 |
| 5.2.3 | adder_dcop_32 | 36 |
| 5.2.4 | dc1 | 38 |
| 5.2.5 | cavity10 | 39 |
| 5.2.6 | mcfe | 40 |
| 5.2.7 | af23560 | 41 |
| 5.2.8 | raefsky2 | 42 |
| 5.2.9 | ML_Laplace | 44 |

| | | |
|----------|--|-----------|
| 5.2.10 | ELLPACK | 45 |
| 5.2.11 | adder_dcop_32 && dc1 | 45 |
| 5.2.12 | cavity10 && mcfe | 46 |
| 5.2.13 | raefsky2 | 47 |
| 5.2.14 | ML_Laplace | 48 |
| 5.2.15 | af23560 | 48 |
| 5.2.16 | Prestazioni con l'utilizzo della cache | 49 |
| 6 | Suddivisione del lavoro | 50 |
| 7 | Istruzioni per la compilazione | 50 |
| 8 | Struttura del codice e sua riutilizzabilità | 54 |

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 siano tutti 1) 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 nel formato *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 le 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 sulla macchina.

Nell'analisi del prodotto parallelo con OpenMP utilizzando il formato CSR possiamo soffermarci a ragionare sui seguenti due aspetti:

- *False Cache Sharing*
- *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:

| | | | | | | |
|------|------|------|------|--|--|-------|
| as: | as1 | as2 | as3 | | | asNZ |
| ja: | ja1 | ja2 | ja3 | | | jaNZ: |
| irp: | irp1 | irp2 | irp3 | | | irpM |

Figure 1: Strutture dati CSR

Inoltre, la funzione `create_dense_matrix()` crea una matrice densa X di dimensione $N \times K$ strutturata nel seguente modo:

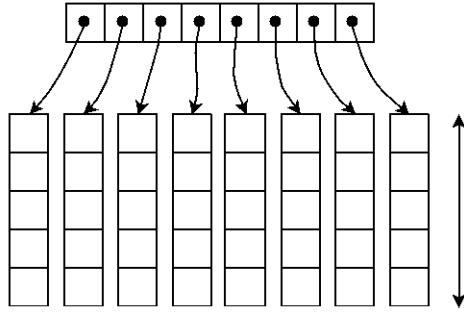


Figure 2: Matrice Densa

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

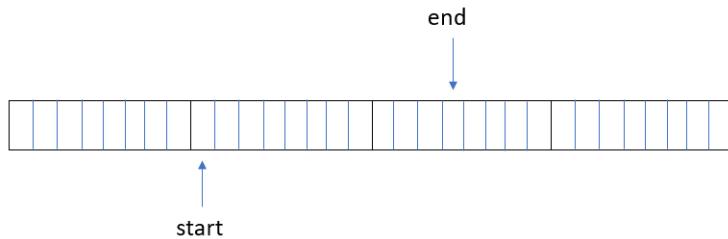
| | | | | | | | | | |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| X11 | ... | XN1 | X12 | ... | XN2 | ... | X1K | ... | XNK |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|

Figure 3: Rappresentazione in memoria lineare della trasposta della matrice X

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 thread 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. 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:



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. Poiché le righe che vengono assegnate ai thread sono consecutive tra di loro 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 . Nel codice vengono effettuate le scritture sulla matrice Y da parte dei vari threads. Poiché gli elementi delle righe della matrice Y sono contigui in memoria e un singolo thread si occupa

degli elementi di una stessa riga, allora si riesce a ridurre il *False Cache Sharing* e a sfruttare la località dei dati.

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 interesse. A seconda della modalità di compilazione (e.g., l'algoritmo che si vuole utilizzare) vengono definite specifiche variabili che saranno successivamente utilizzate 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. Infine, viene deallocata la memoria sia sulla GPU che sull'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 Y_{iz} della matrice finale Y . Il numero di thread 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:

$$\begin{aligned} row &= tid / K \\ col &= tid \% K \end{aligned}$$

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 thread

maggiori 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 continuati 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 versioni 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 una 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 tramite un intero warp o sub-warp ma quella di parallelizzare il calcolo dei K elementi che fanno parte di una stessa riga della matrice Y sfruttando un intero warp. Si è deciso di implementare questa versione anche per il fatto che il numero di elementi K che compongono una riga è molto limitato (al più 64).

A questo punto, scendiamo un po' più nel dettaglio dell'algoritmo. Ogni warp, suddiviso in 8 sub-warp (ciascuno composto da 4 thread), è responsabile del calcolo di tutte le componenti di una riga della matrice Y . I thread del sub-warp collaborano per calcolare uno stesso elemento della matrice Y la cui riga è determinata dall'identificativo del warp di cui il sub-warp fa parte. Viene eseguita una riduzione in parallelo che coinvolge i thread dello stesso sub-warp e solamente il thread con identificativo pari a θ (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 un'intera riga della matrice Y .

Dai grafici sulle prestazioni che sono stati realizzati abbiamo notato il dominio dal punto di vista delle prestazioni dei due algoritmi *CSR Scalar* e *CSR Vector Sub Warp*. Inoltre, al variare delle matrici e del numero di colonne K non esiste un algoritmo che domina sempre sull'altro. Di conseguenza, abbiamo deciso di realizzare una nostra versione dell'algoritmo *Adaptive* che è stato descritto nei vari paper. Il nostro obiettivo è stato principalmente quello di utilizzare l'algoritmo *CSR Vector Sub Warp* negli scenari in cui l'algoritmo *CSR Scalar* non riesce a comportarsi meglio. A questo punto, vediamo un po' più nel dettaglio come è stato implementato questo algoritmo.

Nel *CSR Scalar* ogni thread ha il compito di computare un singolo elemento della matrice finale Y mentre nel *CSR Vector Sub Warp* il calcolo di un singolo elemento della matrice finale Y è delegato a due thread differenti. Poiché nel *CSR Scalar* un singolo thread si occuperà di computare un elemento Y_{ij} , allora sarebbe meglio che il numero di non-zeri nella riga i -esima della matrice sparsa A fosse relativamente basso. L'idea di questo algoritmo *Adaptive* è quella di determinare empiricamente una threshold THR che sia sufficientemente buona nel distribuire il calcolo degli elementi della matrice finale Y tra i due differenti algoritmi:

- Se il numero di non-zeri è sotto questa threshold allora viene eseguito l'algoritmo *CSR Scalar*
- altrimenti, viene eseguito l'algoritmo *CSR Vector Sub Warp* con l'obiettivo di bilanciare tra i vari thread del subwarp il lavoro per calcolare l'elemento Y_{ij} .

Per quanto riguarda l'algoritmo *CSR Vector Sub Warp* bisogna determinare la dimensione del subwarp che denotiamo con *SUB_WARP_SIZE*. Di conseguenza, ho due differenti valori da ricavare empiricamente:

- *THR*: la threshold relativa al numero di non-zeri che determina l'algoritmo da utilizzare tra il *CSR Scalar* e il *CSR Vector Sub Warp*. La scelta viene

fatta basandosi proprio sul numero di non-zeri presenti nelle singole righe della matrice sparsa A .

- SUB_WARP_SIZE : il numero di thread che compongono un subwarp. Eseguendo delle prove abbiamo visto che il numero migliore di thread che possiamo utilizzare è pari a 2.

A questo punto, bisogna ragionare sul numero totale di thread e sul numero totale di blocchi che sono necessari per eseguire la computazione. Utilizzando il *CSR Scalar* abbiamo bisogno di un singolo thread mentre con il *CSR Vector Sub Warp* necessitiamo di due differenti thread ($SUB_WARP_SIZE = 2$). Poiché il numero di thread che viene utilizzato dipende dall'algoritmo siamo stati costretti ad affrontare i seguenti problemi:

- Se i thread che eseguono il *CSR Scalar* e quelli che eseguono il *CSR Vector Sub Warp* sono presenti all'interno dello stesso blocco, allora un thread qualsiasi nel blocco non potrà ricavare l'elemento da calcolare. Ad esempio, il thread con $TID = 1$ non sa se calcolare il primo elemento della matrice Y insieme al thread con $TID = 0$ oppure calcolare il secondo elemento della matrice poiché il primo viene calcolato solamente dall'altro thread.
- I due thread che si occupano di calcolare uno stesso elemento dovrebbero far parte dello stesso warp in modo da evitare l'utilizzo della sincronizzazione esplicita. Tuttavia, se i thread sono mischiati all'interno di uno stesso blocco allora non possiamo avere questa garanzia. Ad esempio, se i primi 31 thread di un blocco utilizzano *CSR Scalar* e il 32-esimo thread utilizza il *CSR Vector Sub Warp* allora l'altro thread nello stesso subwarp si troverà all'interno di un altro warp e, quindi, necessitiamo della sincronizzazione esplicita.

Per risolvere questi problemi abbiamo deciso di raggruppare i thread nei blocchi a seconda dell'algoritmo che verrà utilizzato. In questo modo, ci ritroviamo ad avere *total_block_scalar* blocchi contenenti i thread che dovranno eseguire *CSR Scalar* e i rimanenti blocchi contenenti i thread che dovranno eseguire *CSR Vector Sub Warp*. Il valore *total_block_scalar* può essere utilizzato come threshold per determinare il tipo di blocco a cui il thread fa parte.
Una volta suddiviso i blocchi in due classi differenti, siamo passati a risolvere il seguente problema:

Come fa un thread a recuperare l'informazione relativa all'elemento della matrice Y che deve computare?

Ad esempio, consideriamo il primo thread che fa parte del primo blocco relativo all'algoritmo *CSR Vector Sub Warp*. Il thread non sa a priori quale sia l'elemento che gli è stato associato poiché dipende da come sono stati distribuiti i vari elementi tra i thread. Di conseguenza, l'algoritmo *CSR Adaptive*

dovrà utilizzare delle strutture dati di supporto mantenenti specifiche informazioni tra cui quelle che consentiranno ai thread nei vari blocchi di recuperare l'elemento di cui si dovranno occupare. E' stata introdotta la struttura dati *struct core_adaptive_personalizzato* composta dai seguenti campi:

- *metadata*: questo campo contiene una serie di informazioni generiche utili per l'esecuzione dell'algoritmo:
 1. Numero totale di blocchi (Scalar + Vector)
 2. Numero totale di threads che utilizzano *CSR Scalar*
 3. Numero totale di threads che utilizzano *CSR Vector Sub Warp*
 4. Numero totale di blocchi relativi al *CSR Scalar*
 5. Numero totale di blocchi relativi al *CSR Vector Sub Warp*
 6. Numero totale di threads impegnati nel calcolo degli elementi della matrice Y
- *items_scalar*: contiene le coppie (i, j) che sono rispettivamente l'indice di riga e l'indice di colonna dell'elemento della matrice Y che dovrà essere calcolato con il *CSR Scalar*.
- *items_vector*: contiene le coppie (i, j) che sono rispettivamente l'indice di riga e l'indice di colonna dell'elemento della matrice Y che dovrà essere calcolato con il *CSR Vector Sub Warp*.

La funzione *csr_adaptive_personalizzato_number_of_blocks* viene eseguita sulla CPU e ha il compito di allocare e popolare le strutture dati di appoggio per il *CSR Adaptive*. Le strutture dati *items_scalar* e *items_vector* hanno un numero di elementi pari al numero di thread impiegati rispettivamente nell'utilizzo del *CSR Scalar* e del *CSR Vector*. Queste strutture dati verranno indicizzate tramite l'identificativo del thread considerando solamente i blocchi relativi all'algoritmo che il thread dovrà utilizzare. La funzione *CSR_Adaptive_personalizzato()* implementa il nostro algoritmo che sfrutta il *CSR Scalar* e il *CSR Vector Sub Warp*. Il numero totale di blocchi relativi all'algoritmo *CSR Scalar* viene utilizzato come threshold per capire la tipologia di blocco a cui il thread corrente appartiene.

Osservazione Negli algoritmi *CSR_Vector_by_row*, *CSR Vector* 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 .

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_elpack_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*. Nella funzione *main* viene invocata la funzione *create_dense_matrix* che crea una matrice densa di valori casuali con *N* righe e *K* colonne:

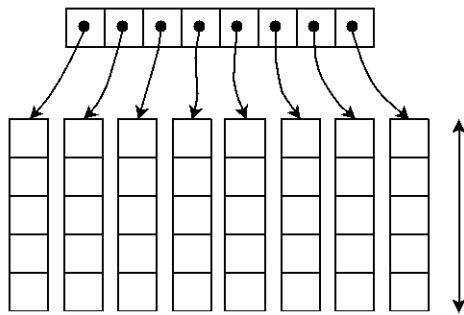


Figure 4: Matrice Densa X

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

| | | | | | | | | | |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| X11 | ... | XN1 | X12 | ... | XN2 | ... | X1K | ... | XNK |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|

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 *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 mentre per quanto riguarda la matrice X utilizziamo la sua trasposta:

| | | | | | | | | | |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| X11 | ... | XN1 | X12 | ... | XN2 | ... | X1K | ... | XNK |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|

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]

Osserviamo che gli elementi appartenenti ad una stessa riga della matrice A 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

- $as[0][j], as[1][j], \dots, as[199][j] \forall j = 0, \dots, max_nz_per_row$
- $aj[0][j], aj[1][j], \dots, aj[199][j] \forall j = 0, \dots, max_nz_per_row$

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:

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

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. 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 effettivamente la responsabilità di determinare le componenti $Y[i][z]$ con $z = 0, \dots, K - 1$.

La differenza tra le due funzioni

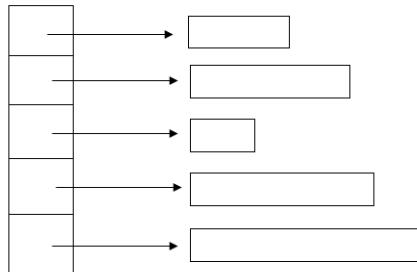
- `parallel_product_ellpack()`
- `parallel_product_ellpack_no_zero_padding()`

sta nella gestione delle righe poiché la versione senza *zero padding* utilizza strutture dati 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 utilizza 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 sono delle matrici "perfette". 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 θ 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.

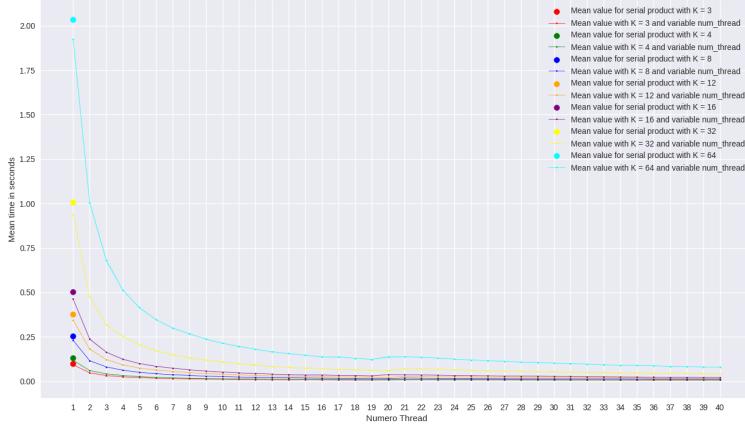
5.1 Prestazioni su CPU

Data la matrice in input, i grafici realizzati su CPU mostrano l'andamento del tempo medio di esecuzione espresso in secondi e dei GFLOPS 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:

Matrice ML_Laplace: Plot della media dei tempi in secondi al variare del numero di threads e K



Matrice ML_Laplace: Plot della media dei GFLOPS al variare del numero di threads e K



I grafici precedenti mostrano l'andamento del tempo medio di esecuzione e dei GFLOPS 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 e i GFLOPS medi 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 della 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:** 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.

La matrice *ML_Laplace* ha un numero medio di non zeri per riga pari a:

$$\frac{27.689.972}{377.002} \approx 73,447$$

La formula utilizzata per calcolare i FLOPS è la seguente:

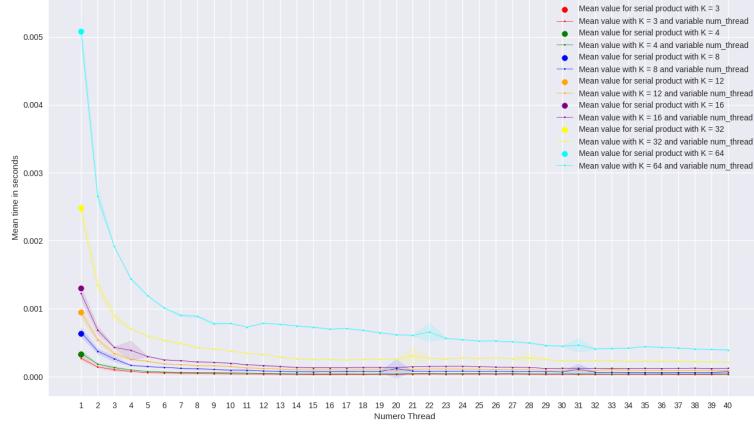
$$FLOPS = \frac{2 \cdot k \cdot NZ}{T}$$

Di conseguenza, essendo il tempo (medio) di esecuzione al denominatore (i.e., inversamente proporzionale ai FLOPS) allora i ragionamenti fatti precedentemente devono essere ribaltati: al crescere del tempo di esecuzione medio i FLOPS diminuiscono e, viceversa, al diminuire del tempo di esecuzione medio i FLOPS aumentano.

Il ragionamento fatto precedentemente è valido anche considerando le altre matrici analizzate nel progetto. I grafici delle prestazioni che mostrano

l'andamento del tempo medio di esecuzione e dei GFLOPS al variare del numero di thread e del numero di colonne K sono mostrati di seguito.

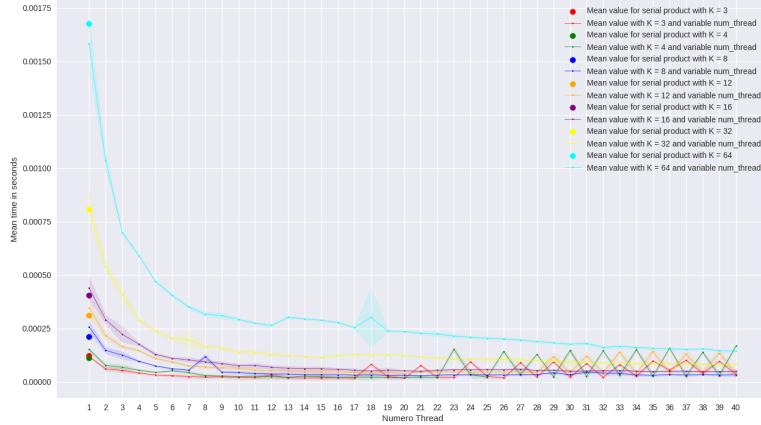
Matrice cavity10: Plot della media dei tempi in secondi al variare del numero di threads e K



Matrice cavity10: Plot della media dei GFLOPS al variare del numero di threads e K



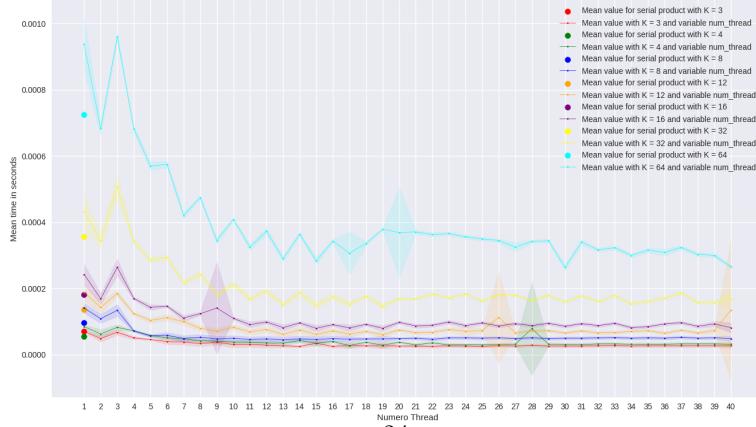
Matrice mcf: Plot della media dei tempi in secondi al variare del numero di threads e K



Matrice mcf: Plot della media dei GFLOPS al variare del numero di threads e K



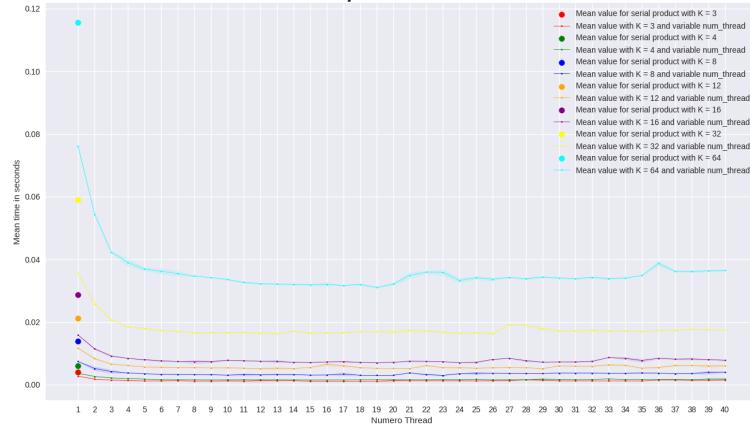
Matrice adder_dcop_32: Plot della media dei tempi in secondi al variare del numero di threads e K



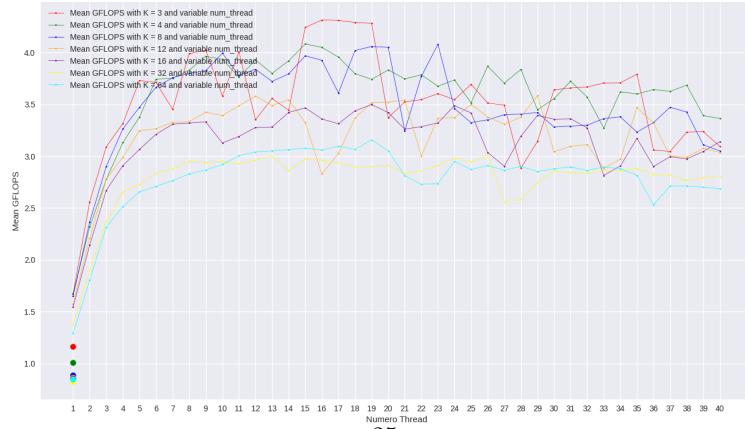
Matrice adder_dcop_32: Plot della media dei GFLOPS al variare del numero di threads e K



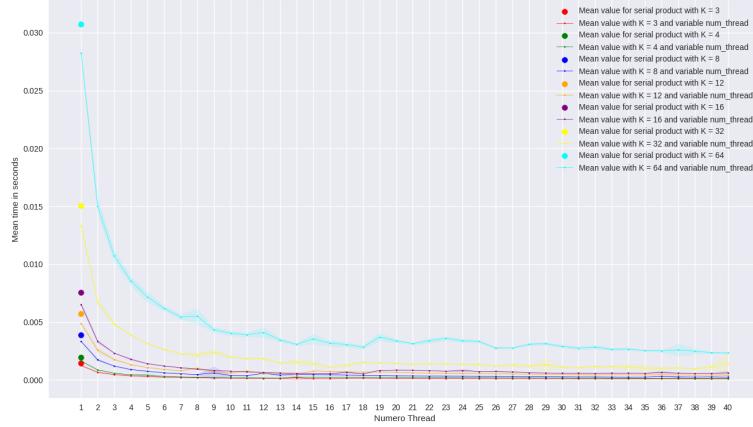
Matrice dc1: Plot della media dei tempi in secondi ai variare del numero di threads e K



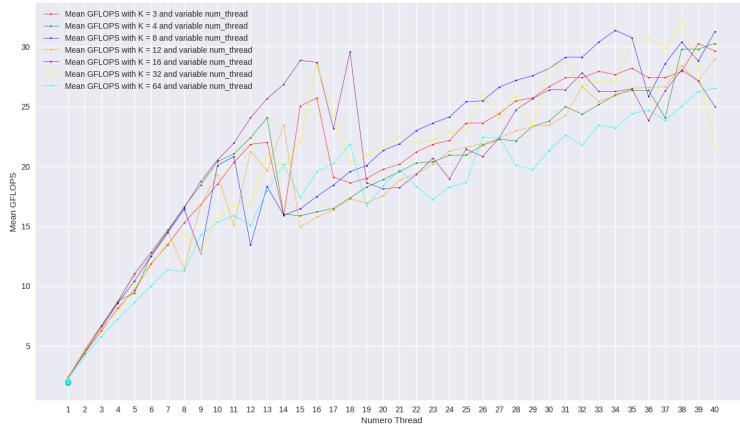
Matrice dc1: Plot della media dei GFLOPS ai variare del numero di threads e K



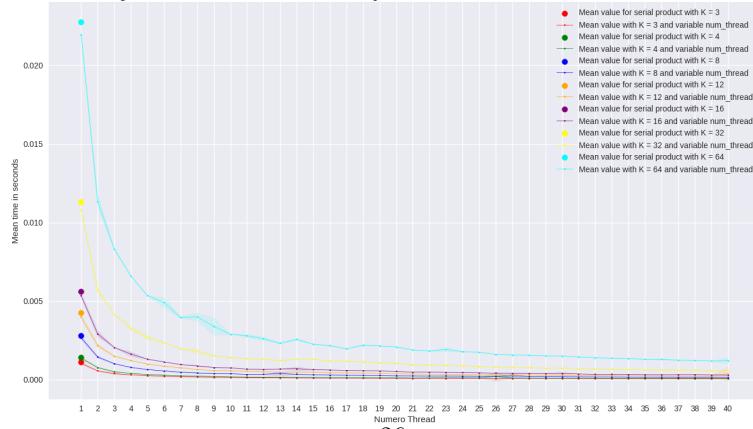
Matrice af23560: Plot della media dei tempi in secondi al variare del numero di threads e K



Matrice af23560: Plot della media dei GFLOPS al variare del numero di threads e K



Matrice rafsky2: Plot della media dei tempi in secondi al variare del numero di threads e K



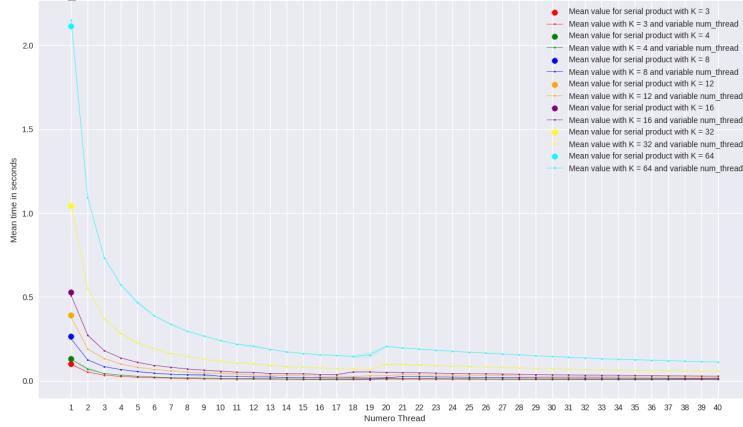
Matrice rafsky2: Plot della media dei GFLOPS al variare del numero di threads e K



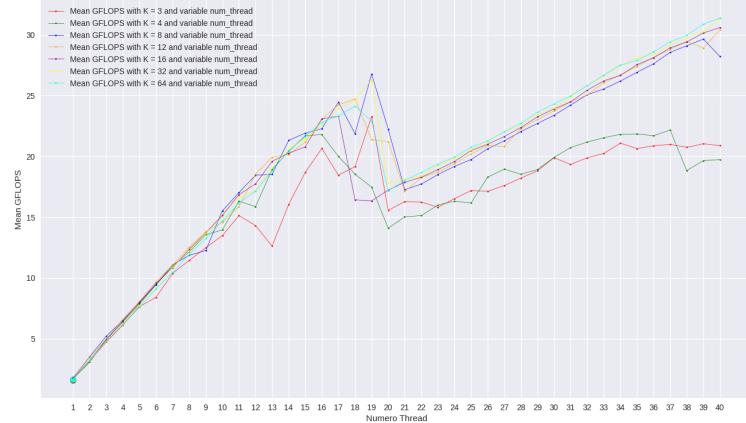
5.1.2 Analisi prestazionale per il formato ELLPACK

Per quanto riguarda l'analisi prestazionale relativa al formato *ELLPACK* possiamo notare un andamento analogo per il tempo medio di esecuzione e per i GFLOPS 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 e dei GFLOPS per alcune delle matrici di test utilizzate nel progetto.

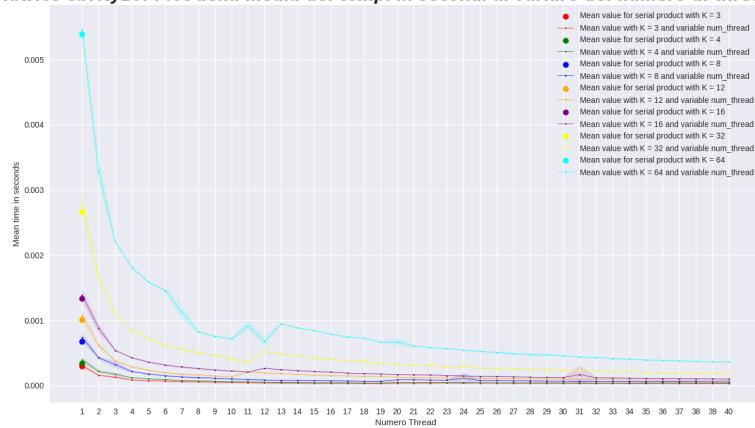
Matrice ML_Laplace: Plot della media dei tempi in secondi al variare del numero di threads e K



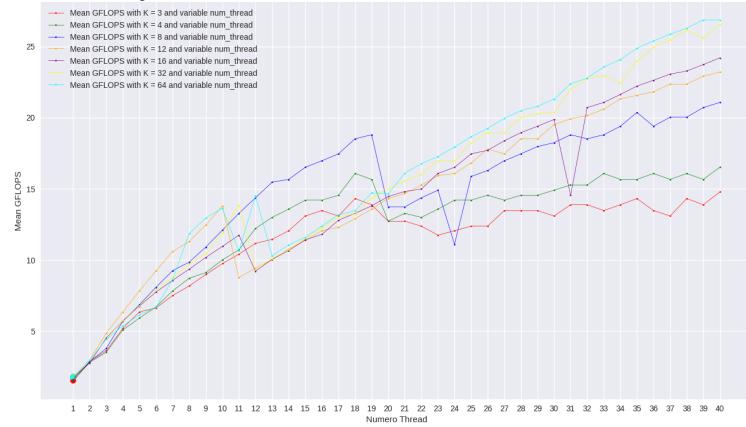
Matrice ML_Laplace: Plot della media dei GFLOPS al variare del numero di threads e K



Matrice cavity10: Plot della media dei tempi in secondi ai variare del numero di threads e K



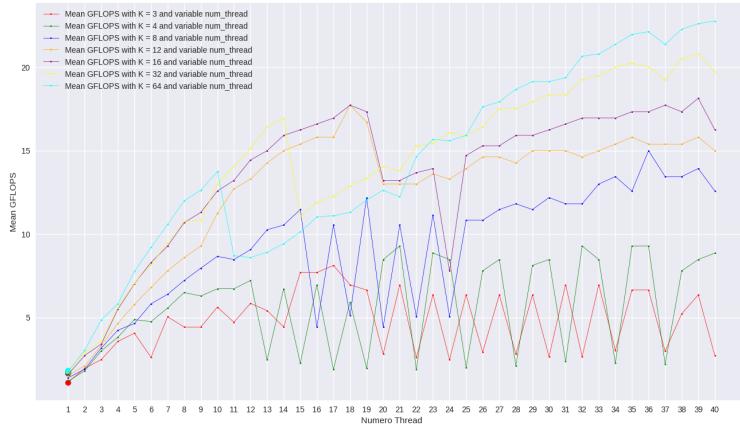
Matrice cavity10: Plot della media dei GFLOPS al variare del numero di threads e K



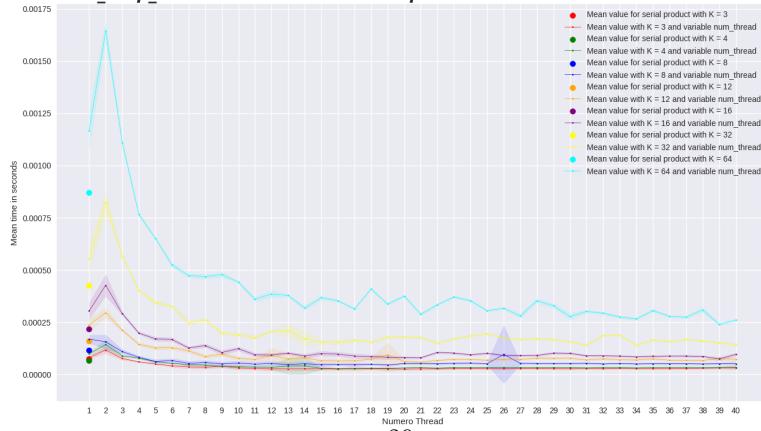
Matrice mcf: Plot della media dei tempi in secondi al variare del numero di threads e K



Matrice mcf: Plot della media dei GFLOPS al variare del numero di threads e K



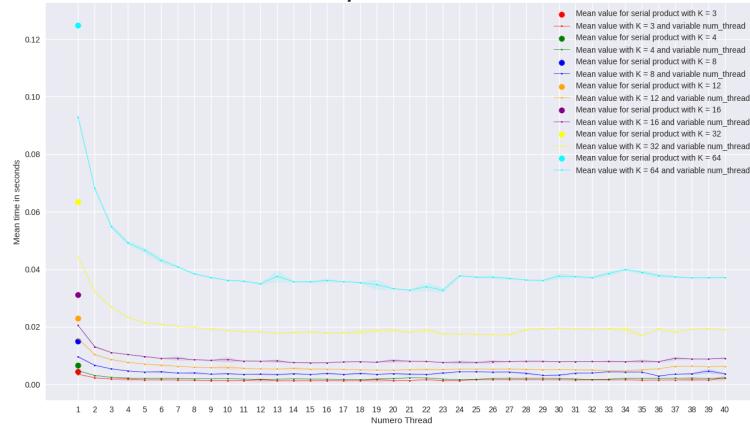
Matrice adder_dcop_32: Plot della media dei tempi in secondi al variare del numero di threads e K



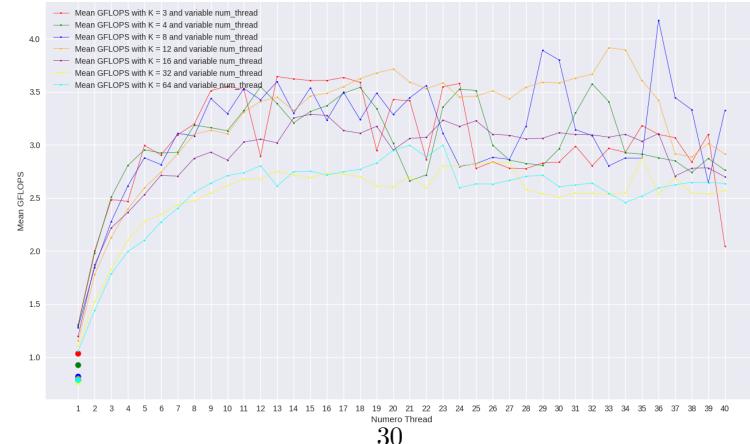
Matrice adder_dcop_32: Plot della media dei GFLOPS al variare del numero di threads e K



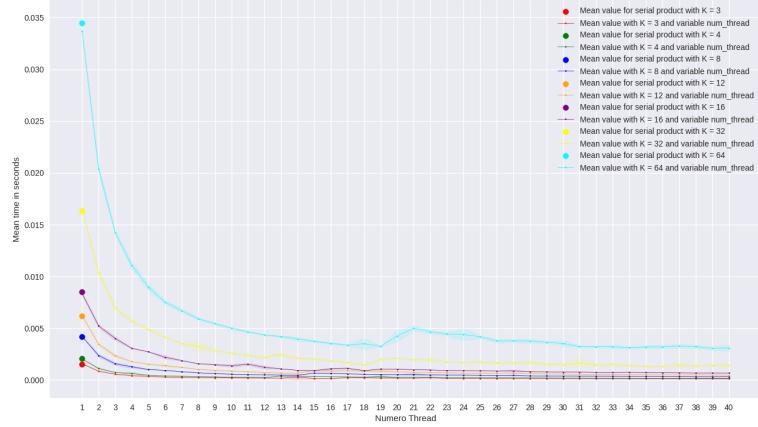
Matrice dc1: Plot della media dei tempi in secondi ai variare del numero di threads e K



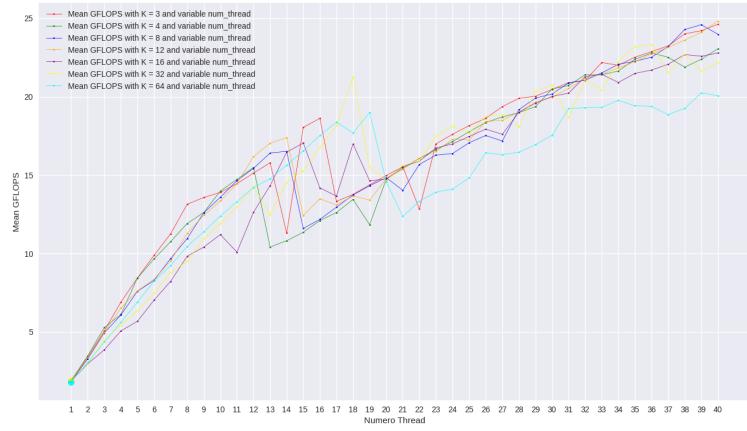
Matrice dc1: Plot della media dei GFLOPS al variare del numero di threads e K



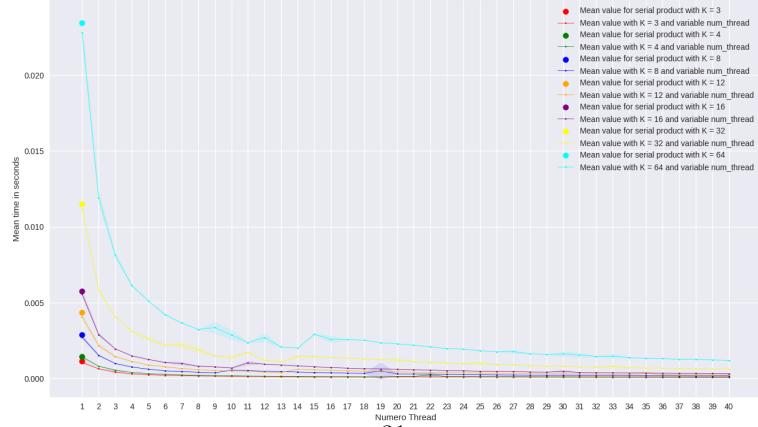
Matrice af23560: Plot della media dei tempi in secondi al variare del numero di threads e K



Matrice af23560: Plot della media dei GFLOPS al variare del numero di threads e K



Matrice rafsky2: Plot della media dei tempi in secondi al variare del numero di threads e K



Matrice raefsky2: Plot della media dei GFLOPS al variare del numero di threads e K



5.2 Prestazioni su GPU

Per quanto riguarda la GPU, abbiamo realizzato dei grafici che mostrano l'andamento medio dei GFLOPS al variare del kernel utilizzato e del numero di colonne K . Inoltre, per studiare meglio le prestazioni degli algoritmi implementati sulle varie matrici, abbiamo realizzato dei file bash che consentono di capire come sono distribuiti i non-zeri nelle varie righe delle matrici. Questi risultati sono stati successivamente graficati per comprendere meglio la distribuzione di interesse.

5.2.1 Campionamento

Per quanto riguarda il campionamento sono state eseguite, in ordine, le seguenti tre strade:

1. *Campionamento senza cache flush*: Le funzioni `samplings_GPU_CSR()` e `samplings_GPU_ELLPACK()` eseguono il campionamento senza svuotare la cache ad ogni lancio del kernel. Infatti, fissata la matrice in input, viene eseguito lo stesso kernel CUDA (algoritmo) per SAMPLING_SIZE volte, al variare del numero di colonne K della matrice X. In questo caso, abbiamo potuto constatare empiricamente, a parità di algoritmo, un aumento delle prestazioni, a causa del mancato cache flush, rispetto alla singola esecuzione dell'intero programma, che è ovviamente con la cache vuota.
2. *Campionamento con `cudaDeviceReset()`*: Le funzioni `samplings_GPU_CSR_dev_res()` e `samplings_GPU_ELLPACK_dev_res()` eseguono il campionamento in modo tale che ogni esecuzione del kernel sia preceduta dall' API `cudaDeviceReset()`. La nostra intenzione era quella di eseguire il flush della cache in modo da avere delle prestazioni più veritieri. Dalla documentazione, sappiamo che la funzione `cudaDeviceReset()` viene utilizzata per il seguente scopo:

Explicitly destroys and cleans up all resources associated with the current device in the current process. Any subsequent API call to this device will reinitialize the device.

Tuttavia, anche con l'utilizzo di questa funzione abbiamo ottenuto delle prestazioni che si discostano molto da quelle ottenute con la singola esecuzione dell'intero programma. Da questo abbiamo dedotto che la cache non viene effettivamente svuotata. In base alle documentazioni trovate in rete non siamo riusciti a trovare il modo per effettuare il flush della cache.

3. *Campionamento automatizzato con singole esecuzioni*: Per far sì che il campionamento non risenta dell'influenza della cache, si è deciso di lanciare il programma più volte al variare della matrice in input, dell'algoritmo utilizzato e del numero di colonne della matrice X. A tal proposito, sono stati realizzati i due script bash `samplings_csr.sh` e `samplings_ellpack.sh` che eseguono quanto detto precedentemente.

Per realizzare i grafici che verranno mostrati di seguito abbiamo utilizzato il terzo metodo tra quelli precedentemente elencati.

5.2.2 CSR

Come detto in precedenza, per il formato CSR abbiamo realizzato differenti kernel. Per analizzare le prestazioni dei kernel conviene confrontare i risultati ottenuti su differenti matrici. Per quanto riguarda le prestazioni, abbiamo tenuto conto dei seguenti aspetti:

- Media dei *non-zeroi* per riga: se la media dei non-zeroi per riga è alta, allora è più conveniente distribuire il carico di lavoro su più thread per il calcolo del singolo elemento della matrice Y .
- Varianza dei *non-zeroi* per riga: se la varianza dei non-zeroi è alta allora tipicamente risulta essere più conveniente utilizzare un algoritmo adattivo.
- Numero di thread per blocco: empiricamente abbiamo riscontrato che il numero di thread per blocco migliore è molto spesso pari a 512.
- Numero di blocchi: poiché i blocchi vengono assegnati agli SM, non è molto conveniente avere molti blocchi poiché aumenta il costo relativo allo scheduling.

L'algoritmo *CSR Adaptive* utilizza una threshold per stabilire quale algoritmo dovrà essere utilizzato per il calcolo di uno specifico elemento della matrice Y . La scelta del valore della threshold dipende dalla matrice sparsa che viene utilizzata nel prodotto. Per stabilire un buon valore della threshold abbiamo utilizzato dei grafici che ci permettono di capire come i non-zeroi sono distribuiti tra le varie righe della matrice sparsa A .

Per quanto riguarda il confronto tra i due kernel *CSR Vector* e *CSR Vector By Row* possiamo fare le seguenti considerazioni:

- Con l'algoritmo *CSR Vector* abbiamo un numero maggiore di thread che cooperano per calcolare uno stesso elemento della matrice Y . Di conseguenza, il numero di accessi in memoria condivisa nell'algoritmo *CSR Vector* sono circa nove volte quelli che abbiamo nell'algoritmo *CSR Vector By Row*.
- Con l'algoritmo *CSR Vector* abbiamo circa K volte il numero dei blocchi che utilizziamo con l'algoritmo *CSR Vector By Row*.

Come mostrato nei grafici di seguito, l'algoritmo *CSR Vector By Row* si comporta sempre meglio dell'algoritmo *CSR Vector*. Infatti, il numero di non-zeroi nelle righe della matrice sparsa A non è sufficientemente grande da mitigare il costo che si paga nell'accedere alla memoria condivisa. Tuttavia, possiamo notare delle eccezioni con le matrici *dc1* e *adder_dcop_32*. Più precisamente:

- Per quanto riguarda la matrice *adder_dcop_32*, l'algoritmo *CSR Vector By Row* si comporta similmente all'algoritmo *CSR Vector*. Infatti, abbiamo una riga che ha un elevato numero di non-zeri rispetto alla media. In questo caso, la distribuzione del carico di lavoro tra i 32 thread del warp riesce a mitigare l'elevato numero di accessi in memoria condivisa.
- Per quanto riguarda la matrice *dc1*, l'algoritmo *CSR Vector* si comporta molto meglio rispetto all'algoritmo *CSR Vector By Row*. Infatti, abbiamo una riga che ha un elevatissimo numero di non-zeri rispetto alla media. In questo caso, la distribuzione del carico di lavoro tra i 32 thread del warp riesce a mitigare notevolmente l'elevato numero di accessi in memoria condivisa.

Nel caso della matrice *adder_dcop_32*, al crescere del numero di colonne K, le prestazioni dell'algoritmo *CSR Vector* si riducono rispetto a quelle dell'algoritmo *CSR Vector By Row*. Al contrario, nel caso della matrice *dc1*, l'algoritmo *CSR Vector* riesce ad avere delle prestazioni migliori. Il motivo di ciò è rappresentato dal fatto che l'outlier della matrice *dc1* ha un numero di non-zeri veramente molto elevato. Di conseguenza, al crescere del numero di colonne K, sebbene le righe con pochi non-zeri che sono coinvolte nel calcolo degli elementi della matrice Y vengono utilizzate più volte, comunque il solo fatto che la riga con tutti quei non-zeri venga utilizzata più volte riesce a mantenere alte le prestazioni. Dai grafici mostrati di seguito, capiamo che l'utilizzo della memoria condivisa comporta spesso un decremento delle prestazioni. Infatti, al diminuire del numero degli accessi alla memoria condivisa, crescono le prestazioni degli algoritmi. Infatti:

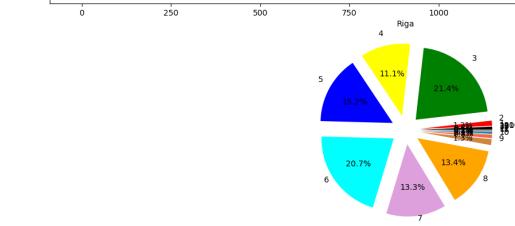
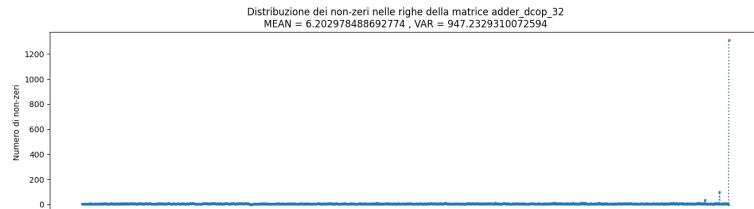
- L'algoritmo *CSR Vector* esegue il maggior numero di accessi alla memoria e si comporta nel complesso peggio degli altri algoritmi.
- Gli algoritmi *CSR Vector By Row* e *CSR Vector Sub Warp* eseguono un numero minore di accessi alla memoria condivisa e hanno delle prestazioni migliori.
- L'algoritmo *CSR Scalar* non utilizza la memoria condivisa e tipicamente ha le prestazioni migliori.
- Infine, l'algoritmo *CSR Adaptive* cerca di risolvere il problema del *CSR Scalar* utilizzando la memoria condivisa nel parallelizzare il calcolo solamente quando il numero di non-zeri è molto elevato (sopra la threshold).

In breve, ci ritroviamo tipicamente le seguenti relazioni per quanto riguarda il numero degli accessi in memoria condivisa:

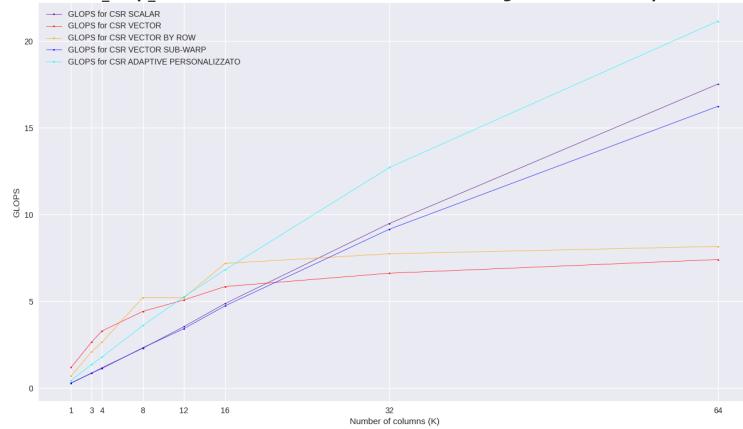
$$\textit{CSR Scalar} > \textit{CSR Vector Sub Warp} > \textit{CSR Vector By Row} > \textit{CSR Vector}$$

Per quanto riguarda l'algoritmo *CSR Adaptive*, esso si pone tra il *CSR Scalar* e il *CSR Vector Sub Warp*.

5.2.3 adder_dcop_32



Matrice adder_dcop_32: Plot dei GFLOPS al variare di K e dell'algoritmo utilizzato per il formato CSR



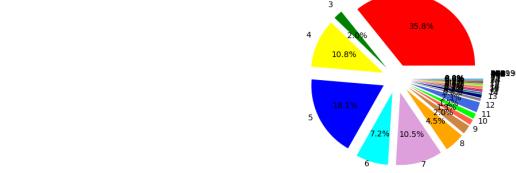
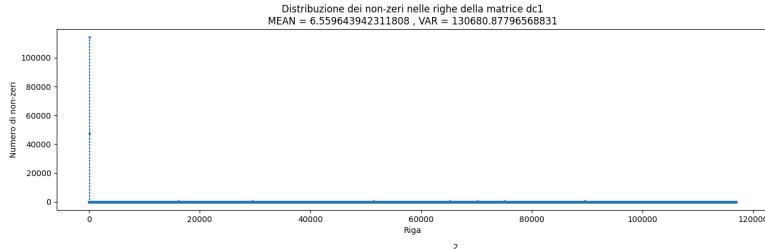
E' possibile fare le seguenti considerazioni sull'applicazione dei kernel alla matrice *adder_dcop_32*:

- La media di non-zeri per riga è molto bassa. Come si può osservare dallo scatter plot, ci sono solamente tre righe che hanno un numero di non-zeri che si discosta dal valore medio:

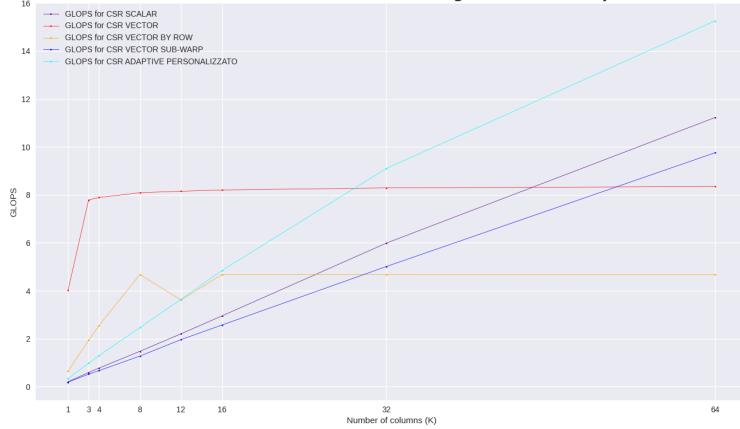
1. 35
2. 100
3. 1310

- Gli outlier elencati in precedenza comportano un elevato valore per la varianza.
- Poiché il numero di non-zeri medio è basso, il *CSR Scalar* ha delle prestazioni migliori rispetto al *CSR Vector Sub Warp* poiché il lavoro da svolgere per calcolare i singoli elementi della matrice Y non è eccessivamente pesante. Tuttavia, bisogna osservare come esistano delle righe della matrice sparsa per le quali il numero di non-zeri è più elevato. In questo caso, l'utilizzo di due thread per il calcolo dell'elemento della matrice Y risulta essere più appropriato. Il motivo per cui il *CSR Scalar* va nel complesso meglio del *CSR Vector Sub Warp* è dovuto al fatto che sono solamente tre gli outlier. L'utilizzo dell'algoritmo *CSR Adaptive* risolve il problema dello *Scalar* per quanto riguarda le righe della matrice sparsa A che hanno un gran numero di non-zeri. Infatti, avendo definito una threshold pari a 30, avremo che tutte le righe della matrice A , ad eccezione delle tre sopra elencate, saranno associate al *CSR Scalar* mentre per il calcolo degli elementi della matrice Y che coinvolgono le righe con molti non-zeri della matrice A verrà utilizzato il *CSR Vector Sub Warp*.

5.2.4 dc1



Matrice dc1: Plot dei GFLOPS al variare di K e dell'algoritmo utilizzato per il formato CSR



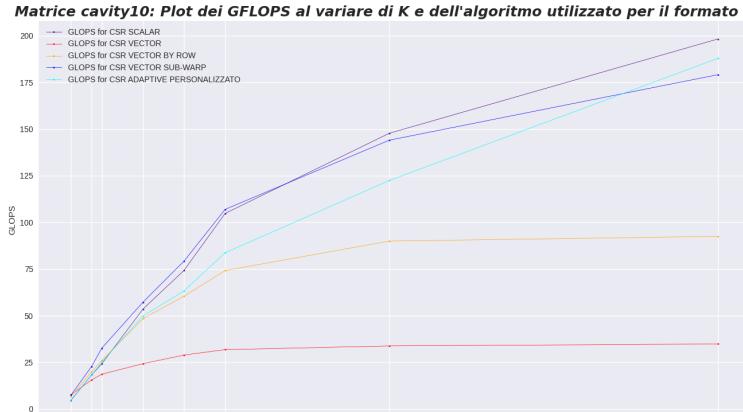
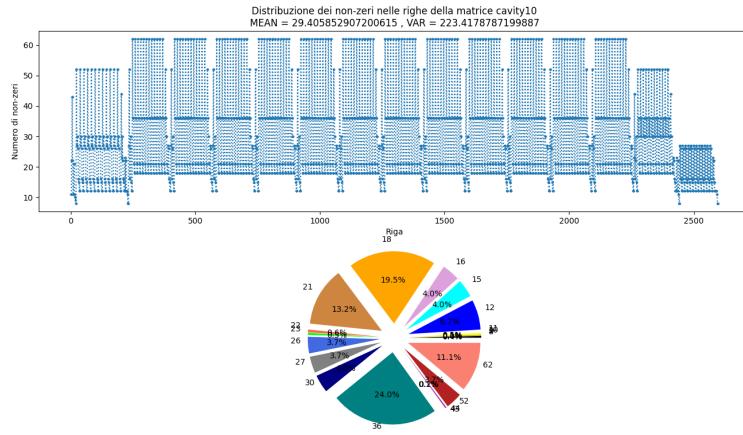
Per quanto riguarda la matrice *dc1*, le osservazioni che possiamo fare sono molto simili a quelle fatte per la matrice precedente. Infatti, il numero medio di non-zeri per riga è basso e la varianza risulta essere molto grande per via della presenza di outlier che hanno un gran numero di non zeri:

1. 114.190
2. 47.193

Come si può osservare dal grafico relativo all’andamento delle prestazioni, l’algoritmo *CSR Adaptive* ha un comportamento migliore poiché parallelizza meglio il cal-

colo degli elementi della matrice Y che coinvolgono le righe con numerosi non-zeri della matrice sparsa A . Con una threshold pari a 50, tutte le righe, ad eccezione delle due elencate di sopra, sono associate al *CSR Scalar*.

5.2.5 cavity10



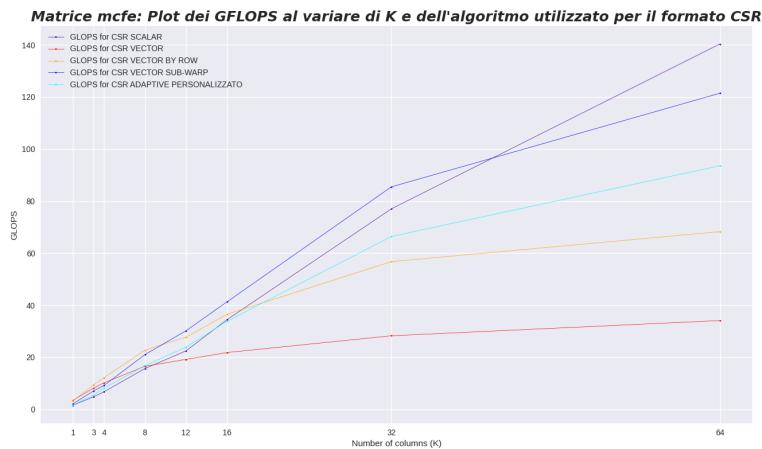
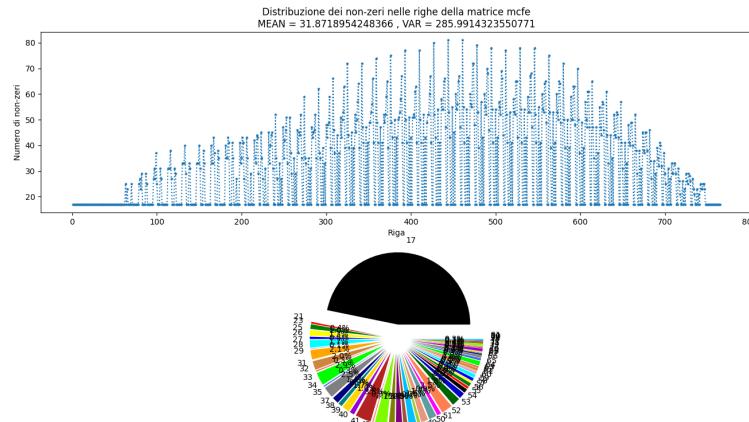
E' possibile fare le seguenti considerazioni sull'applicazione dei kernel alla matrice *cavity10*:

- La media dei non-zeri per riga è maggiore rispetto a quella vista per le matrici precedenti. Una conseguenza di questo fatto è che l'algoritmo

CSR Vector Sub Warp si avvicina all'algoritmo *CSR Scalar* per quanto riguarda le prestazioni.

- La varianza dei non-zeri per riga è minore rispetto a quella vista per le matrici precedenti.

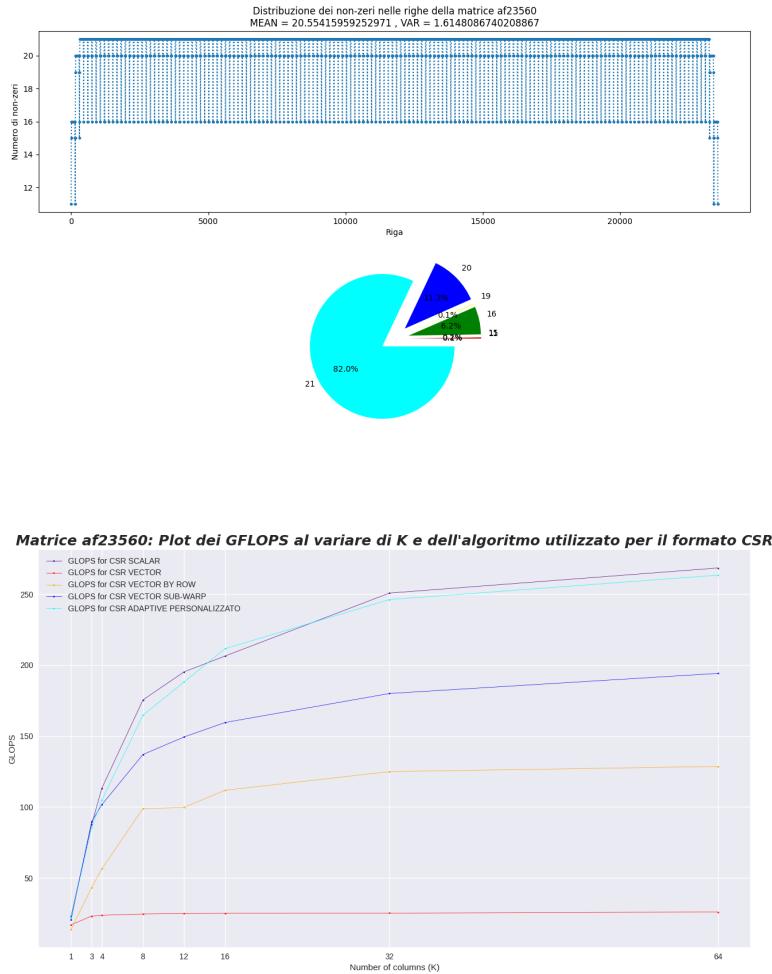
5.2.6 mcf



Per quanto riguarda la matrice *mcf*, le osservazioni che possiamo fare sono molto simili a quelle fatte per la matrice precedente. Infatti, la media e la varianza del numero di non-zeri per riga risultano essere circa la stesse. Inoltre, osserviamo dal grafico riportante l'andamento delle prestazioni dei kernel che

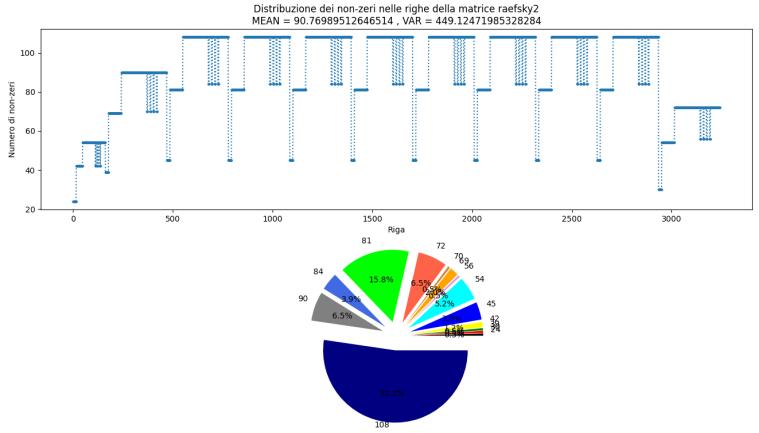
l'algoritmo *CSR Adaptive* non riesce ad avere un risultato nettamente migliore rispetto a *CSR Scalar* e *CSR Vector Sub Warp*.

5.2.7 af23560



Dal grafico rappresentante le prestazioni possiamo notare che l'algoritmo *CSR Vector Sub Warp* va sempre peggio rispetto all'algoritmo *CSR Scalar*. Probabilmente, tutto ciò accade poiché il numero di non-zeri per riga è troppo basso per compensare gli accessi in memoria condivisa e, quindi, è preferibile utilizzare un singolo thread per calcolare gli elementi della matrice Y .

5.2.8 raeftsky2

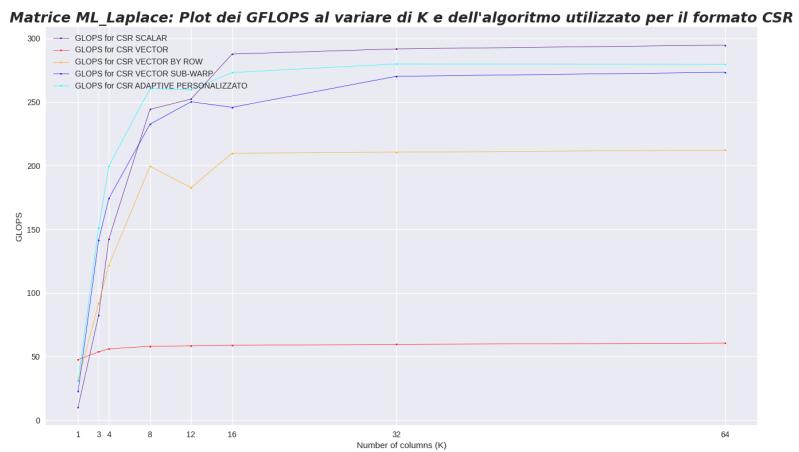
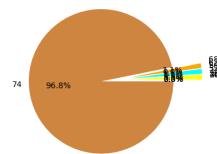
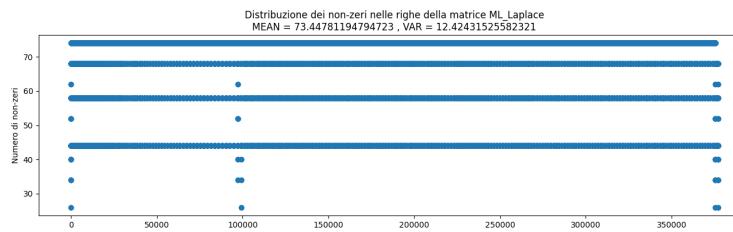


sopra della media. Invece, nella matrice *cavity10* contribuisco alla varianza *anche* un numero elevato di righe che hanno un numero di non-zeri che va al di sotto della media. Tutto ciò, comporta che l'algoritmo *CSR Vector Sub Warp* si comporti meglio con la matrice *raefsky2* rispetto alla matrice *cavity10*.

A valle delle precedenti osservazioni, possiamo osservare che *CSR Vector Sub Warp* si comporta meglio rispetto a quanto avviene con la matrice *cavity10* anche se l'algoritmo *CSR Scalar* riesce ancora ad avere la meglio. Infatti, sebbene gli outlier contengano un numero di non-zeri che va al di sopra del valore medio, tale numero non è eccessivo. Di conseguenza, il costo che si paga utilizzando la memoria condivisa con l'algoritmo *CSR Vector Sub Warp* è ancora dominante nel costo totale della computazione rispetto alla distribuzione del carico di lavoro su più thread.

A conferma di ciò, possiamo notare come l'algoritmo *CSR Vector* tradizionale non riesce ad avere delle buone prestazioni al crescere del numero di colonne K .

5.2.9 ML_Laplace



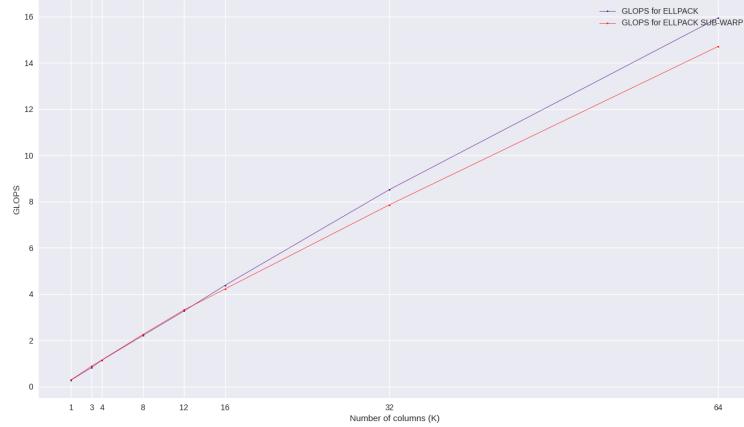
Come si può osservare dal grafico sulla distribuzione di non-zeri, il fatto che la media dei non-zeri per riga sia elevata porta a delle buone prestazioni per l'algoritmo *CSR Vector Sub Warp*. Inoltre, l'algoritmo *CSR Adaptive* riesce a combinare abbastanza bene gli altri due algoritmi ottenendo delle buone prestazioni.

5.2.10 ELLPACK

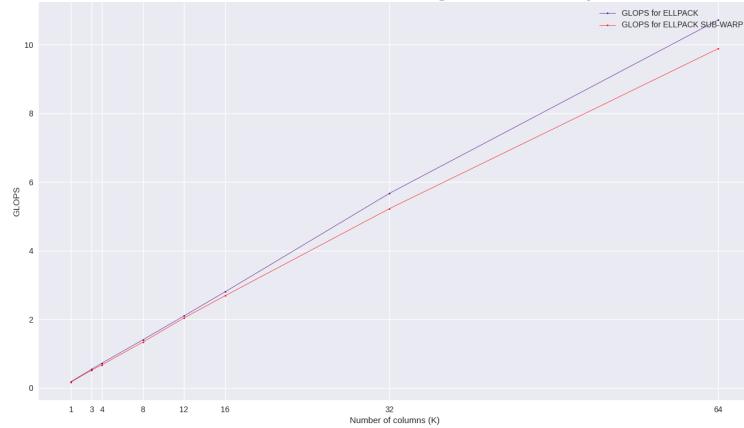
Come detto in precedenza, per il formato ELLPACK abbiamo realizzato differenti kernel. Questi kernel sfruttano la versione ottimizzata del formato ELLPACK poiché la memoria è una risorsa critica sulla GPU. Con l'utilizzo della versione ottimizzata riusciamo a risparmiare una grande quantità di spazio. Per analizzare le prestazioni dei kernel conviene confrontare i risultati ottenuti su differenti matrici. Per quanto riguarda le prestazioni, abbiamo tenuto conto degli stessi aspetti relativi al formato CSR.

5.2.11 adder_dcop_32 && dc1

Matrice adder_dcop_32: Plot dei GFLOPS al variare di K e dell'algoritmo utilizzato per il formato ELLPACK



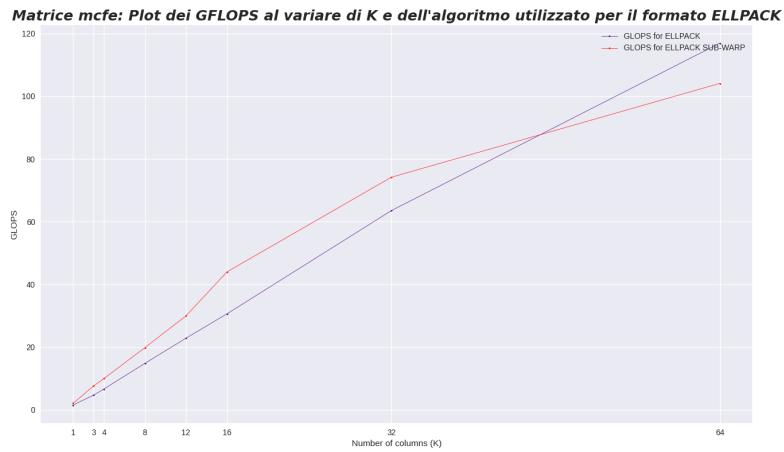
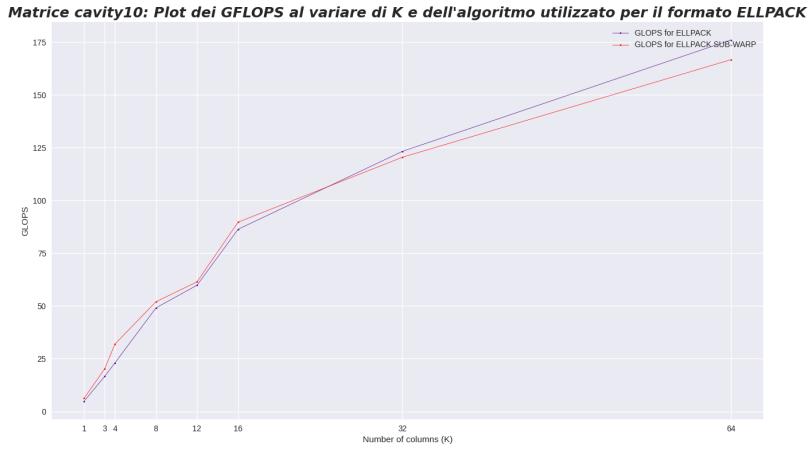
Matrice dc1: Plot dei GFLOPS al variare di K e dell'algoritmo utilizzato per il formato ELLPACK



Come si può osservare dai grafici, l'andamento delle prestazioni dei due algoritmi sono molto simili per le matrici *adder_dcop_32* e *dc1*. Infatti, come possiamo notare dai grafici riportanti la distribuzione dei non-zeri nelle varie righe delle matrici, le due matrici sotto analisi sono molto simili tra di loro:

- Hanno un numero medio di non-zeri basso
- Hanno pochi outlier con un elevato numero di non-zeri

5.2.12 cavity10 && mcfe

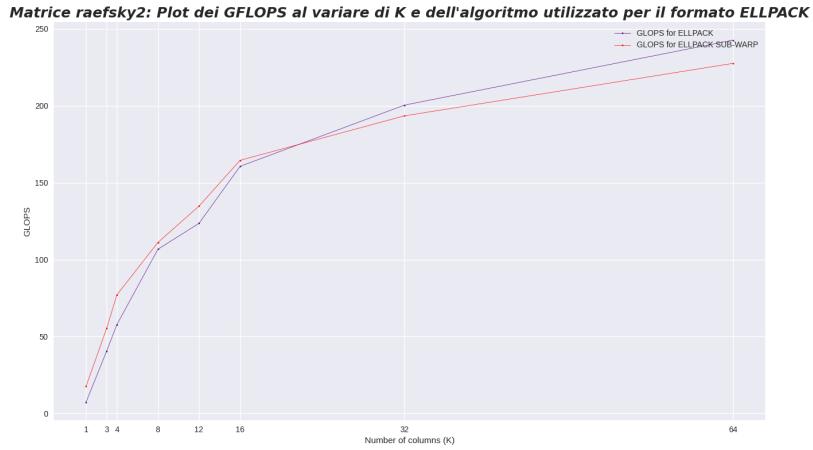


Come si può osservare dai grafici di sopra, l'algoritmo *ELLPACK_Sub_Warp* si comporta quasi sempre meglio rispetto all'algoritmo *ELLPACK*. Infatti, come

possiamo notare dai grafici riportanti la distribuzione dei non-zeri nelle varie righe delle matrici, le due matrici sotto analisi sono molto simili tra di loro:

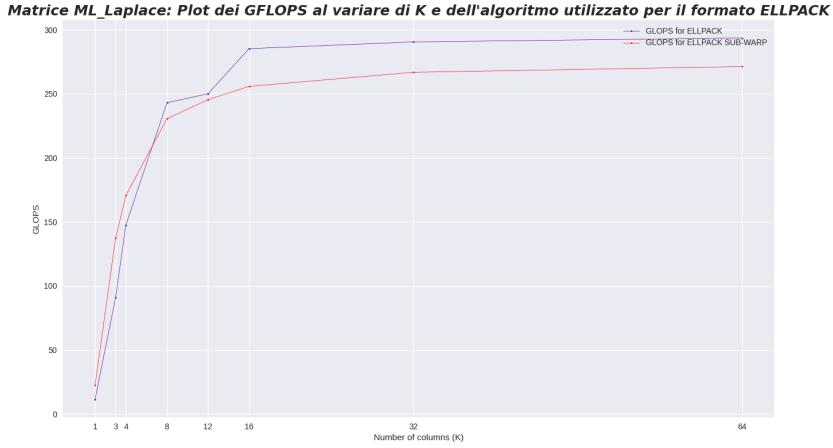
- Hanno un buon numero medio di non-zeri
- La maggior parte delle righe che si discostano dalla media hanno un numero di non-zeri maggiore rispetto alla media. Di conseguenza, l'utilizzo di due thread consente di parallelizzare meglio il lavoro.

5.2.13 raeafsky2



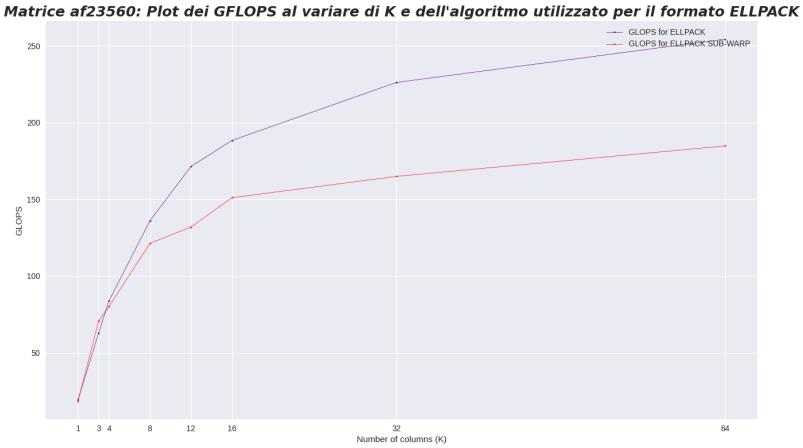
Similmente a quanto detto per le due matrici precedenti, anche per la matrice *raefsky2* possiamo fare le stesse considerazioni. Le uniche differenze sono date dal fatto che il numero medio e la varianza dei non-zeri risultano essere più elevate.

5.2.14 ML_Laplace



Come detto precedentemente per il formato CSR, l'andamento delle prestazioni è molto simile a quanto accade per la matrice *raefsky2*. Per alcuni valori di K l'algoritmo *ELLPACK_Sub_Warp* risulta essere migliore rispetto all'algoritmo *ELLPACK*.

5.2.15 af23560



Come si può osservare dal grafico di sopra, l'algoritmo *ELLPACK* si comporta decisamente meglio. Infatti, il numero medio di non-zeri per riga nella matrice è inferiore rispetto alle matrici precedenti e la varianza è molto bassa. Di conseguenza, non ci sono delle righe con un numero di non-zeri molto maggiore rispetto al valore medio. L'eventuale vantaggio che si crea nel partizionare il carico di lavoro tra due thread lo si perde con l'utilizzo della memoria condivisa utilizzata per implementare la riduzione parallela.

Osservazione Conclusiva Come possiamo notare dal confronto dei grafici ottenuti con i formati CSR e ELLPACK, i casi in cui il *CSR Vector Sub Warp* si comporta meglio (o comunque alla pari) dell'algoritmo *CSR Scalar* corrispondono ai casi in cui l'algoritmo *ELLPACK Sub Warp* si comporta meglio (o comunque alla pari) dell'algoritmo *ELLPACK*.

5.2.16 Prestazioni con l'utilizzo della cache

In questa sezione abbiamo voluto graficare l'andamento delle prestazioni dei kernel sviluppati per il formato CSR per mostrare come la cache può falsare le prestazioni. Come si può osservare dal grafico di seguito, l'aiuto che fornisce la cache porta ad un incremento notevole delle prestazioni dei kernel sulla matrice *ML_Laplace*. Infatti, siamo passati dai quasi 300 GFLOPS che avevamo osservato precedentemente per $K=64$ ai circa 350 GFLOPS mostrati nel grafico successivo. Ovviamente, questo discorso vale anche per le altre matrici considerate nel progetto.



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**

```
1      #!/bin/bash
2      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**

```
1      #!/bin/bash
2      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**

```
1      #!/bin/bash
2      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**

```
1      #!/bin/bash
2      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**

```
1      #!/bin/bash
2      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**

```
1      #!/bin/bash
2      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**

```
1      #!/bin/bash
2      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**

```
1      #!/bin/bash
2      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**

```
1      #!/bin/bash
2      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**

```
1      #!/bin/bash
2      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**

```
1      #!/bin/bash
2      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**

```
1      #!/bin/bash
2      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**

```
1      #!/bin/bash
2      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**

```
1      #!/bin/bash
2      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:

```
1      #!/bin/bash
2      make ML_Laplace 64
```

In questo caso viene avviato il programma con in input la matrice *ML_Laplace* e un numero di colonne della matrice X pari a 64.

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. Sono presenti due varianti per ogni formato. Le funzioni *samplings_GPU_CSR* e *samplings_GPU_ELLPACK* eseguono il campionamento senza svuotare la cache ad ogni lancio del kernel. Invece, le funzioni *samplings_GPU_CSR_dev_res* e *samplings_GPU_ELLPACK_dev_res* eseguono il campionamento in modo tale che ogni esecuzione del kernel sia preceduta dall' API *cudaDeviceReset()*.
- *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.
- *doc*: E' la directory che contiene la relazione e le immagini utilizzate per realizzarla.
- *distribuzioni*: In questa directory sono presenti gli script bash (e le immagini viste in precedenza) per la realizzazione dei grafici sulle distribuzioni dei non-zeri per riga.
- *include*: E' la directory che contiene gli header files.
- *lib*: E' la directory che contiene la libreria ausiliaria *mmio.c*
- *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.
- *samplings_csr.sh* e *samplings_ellpack.sh*: Sono due script bash per eseguire il campionamento delle prestazioni al variare della matrice in input, dell'algoritmo utilizzato e del numero di colonne della matrice X.