

# Sistemi di calcolo parallelo e Applicazioni

## Relazione progetto finale

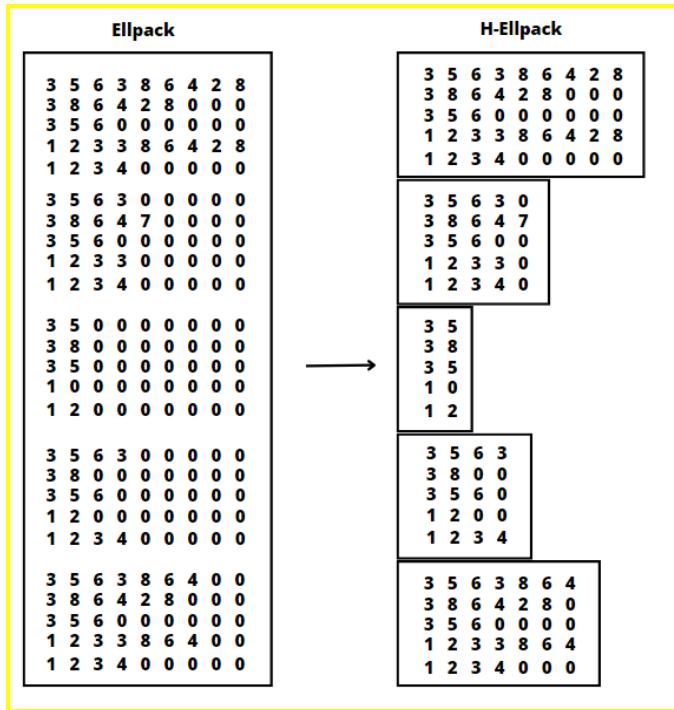
Federica Villani

Mattia Antonangeli

### 1 . Introduzione

Il progetto si pone l'obiettivo di realizzare un nucleo di calcolo per il prodotto tra una matrice sparsa, ovvero una matrice contenente la maggiorparte di elementi pari a zero, e un multivettore, ovvero una matrice con numero di righe molto maggiore rispetto al numero di colonne. Per la rappresentazione in memoria della matrice sparsa, sono stati utilizzati dei formati di memorizzazione creati ad hoc per questa tipologia di matrici, che permettono di salvare in memoria il minor numero di elementi nulli della matrice, evitando, così, di sprecare spazio per la loro memorizzazione. Tali formati sono:

- Compressed Storage by Row (CSR): in questo formato la matrice viene rappresentata mediante l'utilizzo di tre array, ovvero AS, che memorizza gli elementi non zero della matrice, JA, che memorizza le colonne corrispondenti agli elementi non zero della matrice, e IRP, che è un vettore di puntatori all'inizio di ciascuna riga. In questo formato vengono salvati solo i nonzeri della matrice e non abbiamo zeri.
- Ellpack storage format: in questo formato la matrice sparsa viene rappresentata tramite l'utilizzo di due matrici, AS e JA. La prima mantiene il numero di righe della matrice originale e memorizza gli elementi non zero effettuando un padding a zero per le righe che hanno un numero di elementi non zero minore del massimo tra tutte le righe. La seconda memorizza il corrispondente indice di colonna rispetto agli elementi contenuti in AS.
  - H-Ellpack : rivisitazione del formato Ellpack in cui AS e JA vengono divisi in sottomatrici con numero di righe fissato, e ad ogni sottomatrice viene associato un proprio maxnz. In questo modo, tranne per un caso estremo in cui i maxnz sono uguali per ogni matrice, avremo padding minore rispetto a Ellpack



- Coordinate matrix format (COO): questo formato è il più semplice e memorizza per ogni elemento non zero la corrispondente riga e colonna nella matrice originale. Viene utilizzato per il recupero delle matrici sparse dai file .mtx di test.

Per quanto riguarda la generazione e la memorizzazione del multivettore è stata, invece, utilizzata la rappresentazione *Row Major* e, in particolare, una struttura dati contenente due interi che identificano il numero di righe m e colonne n, e un puntatore a un vettore di m\*n elementi.

Per lo sviluppo del nucleo di calcolo che implementa il prodotto tra la matrice sparsa e il multivettore, è stato utilizzato il linguaggio C e sono state definite sia funzioni che eseguono il calcolo in CPU (con OpenMP), sia in GPU (tramite CUDA), entrambe con una versione in CSR e una in ELLPACK.

L'obiettivo del progetto è stato quello di ottimizzare quanto più possibile le prestazioni del nucleo di calcolo creato.

## 2. Sviluppo

Il codice è stato organizzato in due macro-cartelle, *matrices* e *product*. Nella prima sono contenuti: il file *matrixGenerator.h* (con la sua implementazione), relativo alla generazione casuale di multivettori e matrici sparse in rappresentazione *Row Major*; e una cartella *format*, contenente un file per ogni formato di rappresentazione di una matrice sparsa sopra descritti. In ognuno di essi è definita una *struct* per la rappresentazione del formato stesso e delle funzioni di conversione tra formati diversi. E', inoltre, presente un file di libreria che permette di leggere le matrici in formato *MatrixMarket* e di convertirle in formato COO, utile per utilizzare delle matrici di test esterne. La cartella *product*, invece, contiene quattro file,

ognuno dei quali implementa il prodotto in CUDA o in openMP con il formato ELLPACK o CSR. E' presente, inoltre, un file che calcola il prodotto in seriale, utilizzando il formato CSR. Nel *main.c*, infine, vengono raccolte le prestazioni e memorizzate su file csv per poter poi essere analizzate.

## 2.1. CSR

### 2.1.1. CSR seriale

Il prodotto seriale tra una matrice sparsa e un multivettore è stato calcolato nella funzione *calcola\_prodotto\_seriale()* che prende come input un parametro di tipo *csr\_matrix*, corrispondente alla matrice sparsa, uno di tipo *matrix*, corrispondente al multivettore, e un puntatore a *matrix*, in cui viene passato l'indirizzo della struttura dove scrivere il risultato. La sua implementazione è stata necessaria per il confronto delle prestazioni e per verificare la correttezza dei prodotti calcolati in parallelo. Come accennato in precedenza, per questo calcolo è stata utilizzata la rappresentazione della matrice in formato CSR. La funzione scorre il vettore IRP dei puntatori alle righe e, per ogni colonna del multivettore, scorre gli elementi di AS e JA relativi alla riga fissata e calcola il prodotto corrispondente.

### 2.1.2. CSR su CPU

Il prodotto con openMP tra una matrice sparsa e un multivettore è stato calcolato nella funzione *calcola\_prodotto\_per\_righe\_csr\_openmp()* che, oltre a prendere in input gli stessi elementi del csr seriale, ha come parametro il numero di thread da utilizzare nel calcolo. L'idea è stata quella di andare a suddividere le righe della matrice tra i vari thread, in modo che ognuno di essi fosse in grado di calcolare totalmente un certo numero di elementi della matrice risultante. Questo si è tradotto nell'utilizzo della direttiva *#pragma omp for* per suddividere gli elementi del vettore IRP tra i thread. Per evitare problemi di false cache sharing, viene utilizzato lo *schedule static*, quindi lo spazio delle iterazioni viene suddiviso in porzioni di dimensione pari a *chunksize* assegnate staticamente ai thread. Il *chunksize* è stato calcolato in base al numero di righe della matrice e al numero di thread utilizzati in modo tale da renderlo multiplo della dimensione della riga di cache e non arrecare rallentamenti all'esecuzione dovuti al false cache sharing, in cui più thread cercano di accedere alla stessa linea di cache contemporaneamente. Più nello specifico, il *chunksize* viene impostato ad un valore pari al massimo multiplo della dimensione della riga di cache ma minore del quoziente risultante dalla divisione tra il numero di righe della matrice e il numero di thread. Se questo calcolo dà luogo ad un *chunksize* minore della dimensione della riga di cache, viene automaticamente impostato pari alla dimensione della riga di cache.

$$\begin{aligned} \text{chunkSize} &= \max(\text{dimCache}, \max(A)) \\ A &= \{x \in N: x \% \text{dimCache} = 0, x \leq nRighe / nThread\} \end{aligned}$$

Il tempo di esecuzione viene calcolato tramite la struttura dati *clock\_gettime* della libreria *time.h*.

```
#include <smath.h>
performance calcola_prodotto_per_righe_csr_openmp(csr_matrix csrMatrix, matrix multivector,matrix* result, int nThreads){

    if(csrMatrix.n != multivector.m){
        printf("Prodotto non calcolabile tra la matrice e il multivettore inserito\n");
        exit(1);
    }
    double partialSum;
    int i,j,k, irp_1,irp_2;
    int m = csrMatrix.m,n=multivector.n;
    struct timespec start, end;
    int chunkSize = ((int)((csrMatrix.m/(float)nThreads)/16))*16;
    if(chunkSize < 16) chunkSize = 16;
    clock_gettime(CLOCK_MONOTONIC, &start);
    #pragma omp parallel for schedule(static,chunkSize) num_threads(nThreads) firstprivate(n,m) private(partialSum,i,j,k, irp_1, irp_2)
    for(i = 0; i < m; i++){
        irp_1 = csrMatrix.irp[i];
        irp_2 = csrMatrix.irp[i+1];

        for(k = 0; k < n; k++){
            partialSum = 0;
            #pragma omp reduction(+: partialSum)
            for(j = irp_1; j < irp_2; j++){
                partialSum += csrMatrix.as[j]*multivector.coeff[csrMatrix.ja[j]* n + k];
            }
            result->coeff[i * n + k] = partialSum;
        }
    }
    clock_gettime(CLOCK_MONOTONIC, &end);
    performance_stop();
}
```

### 2.1.3. CSR su GPU

Il prodotto con CSR in CUDA è stato calcolato nel file *product\_csr\_GPU.cu* all'interno della directory *product* che prende come input un parametro di tipo *csr\_matrix*, corrispondente alla matrice sparsa, uno di tipo *matrix*, corrispondente al multivettore, e un puntatore a *matrix*, in cui viene passato l'indirizzo della struttura dove scrivere il risultato. Considerando che il formato di rappresentazione CSR classico presenta basse prestazioni su GPU poiché non favorisce gli accessi coalescenti, si è deciso di implementare l'algoritmo CSR-Adaptive. Esso lascia invariata la rappresentazione CSR della matrice e adatta l'esecuzione del calcolo in base alla lunghezza delle righe della matrice. In particolare, infatti, prima di iniziare l'esecuzione sul dispositivo, viene chiamata la funzione *calculate\_rows\_block()*, che, a partire dal vettore IRP e dal numero massimo di elementi non zero che la shared memory di ogni blocco può contenere, calcola il numero di righe contigue che ogni blocco può gestire e ritorna il risultato in un vettore di dimensione pari al numero di blocchi necessari per effettuare il calcolo. La dimensione della griglia è stata impostata pari al numero di blocchi necessari calcolati come appena descritto, mentre la dimensione del blocco è stata fissata al massimo, ovvero 1024. Entrambi sono unidimensionali.

L'algoritmo, implementato nella funzione *csrAdaptiveMultOttimizzato()* eseguita sul dispositivo, sceglie, in base al numero di righe assegnate al blocco corrente, se eseguire CSR Vector, nel caso di una sola riga, o CSR Stream, nel caso di più righe.

In CSR Vector, se gli elementi di AS e JA della riga da gestire sono di un numero tale da poter essere memorizzati nella shared memory, vengono letti dalla memoria globale in maniera coalescente da tutti i thread del blocco e copiati in due vettori, *vals* e *cols*, per poter essere acceduti più velocemente e più volte. I diversi warp del blocco, quindi, accedono agli elementi di *vals* e *cols* se possibile o in maniera coalescente agli elementi del vettore AS e JA, ed eseguono il prodotto con colonne diverse del multivettore in base al loro indice nel blocco (ad esempio se l'indice del warp è 3 e il numero di colonne del multivettore è 64, esso eseguirà il prodotto con la colonna 3 e con la 35). I thread memorizzano, quindi, i risultati

parziali in un array privato di dimensione pari al numero di colonne del multivettore e calcolano, in seguito, tramite la funzione *warp\_reduce()*, il risultato totale. Il primo thread di ogni warp, quindi, si occuperà di copiare il risultato relativo alle colonne del suo warp sulla matrice risultante.

In CSR Stream, invece, siamo già sicuri che gli elementi delle righe da gestire possono essere memorizzate nella shared memory e quindi avviene sempre una copia in due array, *vals* e *cols*, degli elementi di AS e JA dopo una lettura da parte dei thread, sempre tramite accessi coalescenti, dalla memoria globale. Dopodiché, ogni thread prende in carico una riga e una colonna del multivettore, esegue il prodotto tra queste prendendo i valori corrispondenti dagli array nella shared memory e memorizza, poi, il risultato nella matrice risultante.

La funzione che chiama il kernel è *calcola\_prodotto\_csr\_cuda()* ed al suo interno vengono eseguite tutte le *cuda\_malloc*, *cuda\_memcpy* e *cuda\_free* necessarie per il trasferimento dei dati al dispositivo. Il tempo di esecuzione viene calcolato tramite i *cuda\_event* e ritornato come risultato della funzione.

```
global void csrAdaptiveMultOttimizzato(double* as, int* ja, int* irp, double* multivector, int m, int n, int col_multivector, int* rowBlocks, double* resultData){
    shared_double vals[MAX_NNZ_PER_WG];
    shared_int cols[MAX_NNZ_PER_WG];
    int startRow = rowBlocks[blockIdx.x];
    int stopRow = rowBlocks[blockIdx.x+1];
    long int numRows = stopRow - startRow;
    int nnz = irp[stopRow]-irp[startRow];
    int tid = threadIdx.x; // indice del thread nel blocco
    if (numRows > 1){
        //CSR Stream
        int localCol;
        for(int i = tid; i < nnz; i+= blockDim.x){
            localCol = irp[startRow]+i;
            vals[i] = as[localCol];
            cols[i] = ja[localCol];
        }
        int firstCol = irp[startRow];
        __syncthreads();
        for(int t = tid; t < numRows*col_multivector; t += blockDim.x){
            int localRow = startRow + t/col_multivector;
            int j = t%col_multivector;
            double temp = 0;
            for(int i = irp[localRow]-firstCol; i < irp[localRow+1]-firstCol; i++){
                temp += vals[i]*multivector[cols[i]*col_multivector + j];
            }
            resultData[localRow*col_multivector + j] = temp;
        }
        __syncthreads();
    }else{
        //CSR-Vector
        int warpId = tid / 32;
        int lane = tid &(32-1);
        double val;
        int col;
        double sum[64] = {0};
        if(nnz < 4096){
            int localCol;
            for(int i = tid; i < nnz; i+= blockDim.x){
                localCol = irp[startRow]+i;
                vals[i] = as[localCol];
                cols[i] = ja[localCol];
            }
            __syncthreads();
            if(warpId < col_multivector){
                for(int col_m = warpId; col_m < col_multivector; col_m +=32){
                    for(int i = irp[startRow] + lane; i < irp[startRow+1]; i +=32){
                        if(nnz < 4096){
                            val = vals[i-irp[startRow]];
                            col = cols[i-irp[startRow]];
                        }else{
                            val = as[i];
                            col = ja[i];
                        }
                        sum[col_m] += val*multivector[col*col_multivector + col_m];
                    }
                    sum[col_m] = warp_reduce(sum[col_m]);
                    if(lane == 0){
                        resultData[startRow*col_multivector + col_m] = sum[col_m];
                    }
                }
            }
        }
    }
}
```

## 2.2. ELLPACK

In questo nucleo di calcolo viene usato sia il formato ELLPACK già descritto, sia il formato Hacked ELLPACK (H-ELLPACK). Quest'ultimo è una versione di ELLPACK rivisitata al fine di ridurre il più possibile lo spreco di memoria causato dal padding di zeri presente in ELLPACK. La modifica consiste nel suddividere le matrici AS e JA in più matrici il cui numero di righe è fissato (per sfruttare al meglio i warp delle GPU si fissa ad un multiplo di 32), e ogni sottomatrice avrà un proprio numero massimo di nonzeri relativo a quel sottoinsieme di righe.

La gestione e la conversione delle matrici nel formato (H-)ELLPACK è implementata nel file *ellpack.c* presente in *src/matrices/format*. All'interno di questo file troviamo diverse funzioni:

- *convert\_to\_ellpack* e *convert\_coo\_to\_ellpack*: effettua le conversioni di matrici Row Mayor e COO in matrici ELLPACK;
- *convert\_coo\_to\_h\_ellpack*: effettua le conversioni di matrici COO in matrici Hacked ELLPACK;
- funzioni di print su console e su file delle matrici in formato (H-)ELLPACK.

## 2.4. ELLPACK su CPU

Per quanto riguarda il prodotto matrice-multivettore in CPU, esso è implementato avendo una matrice in formato ELLPACK nel file *product\_ellpack\_openmp.c*. In particolare, in questo file troviamo tre funzioni:

- *ellpack\_product*: esegue il prodotto in seriale (non troviamo infatti nessun pragma omp). In particolare, troviamo tre cicli for innestati, in cui il primo cicla sull'indice di riga, il secondo sull'indice di colonna e il terzo sulle colonne del multivettore.
- *omp\_ellpack\_product*: qui avviene l'esecuzione in parallelo in CPU via OpenMP. In particolare, il primo for (visto su *ellpack\_product*) viene parallelizzato tramite la direttiva di omp *#pragma omp for*. In questo modo, ogni thread gestisce il prodotto di una riga della matrice *optimized\_omp\_ellpack\_product*: versione ottimizzata della funzione precedente. La parallelizzazione avviene allo stesso modo, ma, in base ai casi, possono essere effettuate meno operazioni per thread. L'idea è quella di ignorare, nelle somme parziali, i prodotti relativi al padding di zeri che abbiamo per costruzione nelle matrici in formato ELLPACK, in modo da eseguire meno operazioni e quindi ottimizzare il prodotto.

In entrambe le versioni con openMP descritte, viene utilizzato lo *schedule static* e, per evitare il false cache sharing, il chunkSize viene impostato in modo tale che ad ogni thread vengano assegnati degli intervalli di indici che riempiono un'intera linea di cache. In particolare, il chunkSize viene impostato utilizzando la formula già presentata nella trattazione del formato CSR.

```

#pragma omp parallel for schedule(static,chunkSize) num_threads(nThreads) shared(result, mat, vector) firstprivate(m,n,maxnz) private(t,i,j,k,prev_JA, curr_JA)
for (i = 0; i < m; i++) {
    for (int k = 0; k < n; k++) {
        prev_JA = -1;
        t = 0;
        for (int j = 0; j < maxnz; j++) {
            curr_JA = mat.JA[i*maxnz+j];
            if (prev_JA==curr_JA){
                prev_JA = curr_JA;
                t = t + mat.AS[i*maxnz+j]*vector.coeff[curr_JA*n+k];
            }else j = maxnz;
        }
        result->coeff[i*n+k] = t;
    }
}

```

## 2.5. ELLPACK su GPU

Per quanto riguarda il prodotto matrice-multivettore in GPU, sono stati eseguiti diversi tentativi di prodotto:

1. Utilizzo di matrice in formato ELLPACK. Ogni thread esegue il prodotto tra un elemento di una riga della matrice e un elemento di una colonna del multivettore. Questo approccio presenta due problemi principali:
  - a. Per matrici grandi con valori troppo elevati di maxnz, l'esecuzione terminava con errore “out of memory”.
  - b. Accesso non coalescente ai dati: per come è definito il codice, i primi n thread eseguono i prodotti tra la prima riga e le n colonne del multivettore, per cui accedono tutti agli stessi elementi della prima riga, e così via.
2. Utilizzo di matrice in formato H-ELLPACK. Ogni thread esegue il prodotto tra una riga della matrice e una colonna del multivettore. L'utilizzo del formato H-ELLPACK risolve il problema di occupazione di memoria, ma non quello dell'accesso coalescente ai dati.
3. Utilizzo di matrice in formato H-ELLPACK. Ogni warp esegue il prodotto tra una riga della matrice e una colonna del multivettore. Questo approccio risolve il problema dell'accesso coalescente dei dati, ma in caso di maxnz minore di 32 presenta una inefficienza dovuta al fatto che, per ogni sottomatrice, nRow\*(32-maxnz) thread non eseguono operazioni.
4. Utilizzo di matrice in formato H-ELLPACK. Ogni blocco CUDA calcola i prodotti relativi a una sottomatrice AS e JA. Nel caso in cui il numero di sottomatrici sia maggiore del massimo numero di blocchi allocabili, ci potranno essere dei blocchi che gestiscono più sottomatrici. Per sfruttare quanto più possibile la shared memory, si cerca di caricare le sottomatrici AS e JA all'interno della shared memory. Se questo non è possibile a causa delle loro dimensioni troppo elevate, la sottomatrice viene a sua volta suddivisa per righe in sotto-sottomatrici di dimensioni tali da poter essere salvate in shared memory. In questo modo, iterativamente, l'algoritmo sfrutterà tutti i thread del blocco per il caricamento in shared memory della sotto-sottomatrice a partire dalla global memory ed eseguirà il prodotto tra essa e tutte le colonne del multivettore (assegnando ad ogni thread il compito di eseguire il prodotto tra una riga della matrice e una colonna del multivettore), salvando il risultato nella matrice risultante per poi passare alla sotto-sottomatrice successiva. Nel caso in cui la dimensione della riga della sottomatrice sia troppo grande per essere caricata

completamente nella shared memory, il prodotto viene effettuato leggendo gli elementi di AS e JA dalla global memory.

I blocchi sono impostati sempre alla dimensione di 1024 thread, mentre la dimensione della griglia è pari al minimo tra il numero di sottomatrici e il numero massimo di blocchi allocabili nella griglia.

Gli approcci sopra definiti sono stati implementati nel file product\_ellpack\_gpu.cu.

```
__global__ void optimized_cuda_h_ellpack_product(int m, int n, int* maxnz, double* AS, int* JA, int* hackOffsets, int hackSize, int numMatrix, int matDim, double* coeff, double* myRes){
```

```
int bid = blockIdx.x;
int tid = threadIdx.x;

// Definisco vettori in shared memory che conterranno gli elementi delle "sotto-sottomatrici" di AS e JA
__shared__ double vals[4096];
__shared__ int cols[4096];

// Le variabili first e last sono utilizzate per scorrere le sotto-sottomatrici
int first = 0;
int last = 0;

double res=0;

// Primo ciclo for: scorre più sottomatrici per singolo blocco nel caso in cui le sottomatrici sono più del massimo numero dei blocchi
for (int submatIdx = bid; submatIdx < numMatrix; submatIdx += gridDim.x){
    // Se il numero delle colonne di AS e JA supera 4096, non è possibile salvare nemmeno una riga nella shared memory. In questo caso, si esegue il calcolo accedendo in memoria globale.
    if(maxnz[submatIdx] < 4096){
        int counter = 0;
        //Secondo ciclo for: scorre le righe delle "sotto-sottomatrici"
        for (first = hackOffsets[submatIdx]; first < hackOffsets[submatIdx + 1]; first += (int)((4096.0/maxnz[submatIdx])*maxnz[submatIdx])){first += (int)((4096.0/maxnz[submatIdx])*maxnz[submatIdx])};
        last = min(first + (int)((4096.0/maxnz[submatIdx])*maxnz[submatIdx]), hackOffsets[submatIdx + 1]);
        //Terzo ciclo for: salva i valori di AS e JA nella memoria condivisa sfruttando tutti i thread del blocco
        for (int idxS = first + tid; idxS < last; idxS += blockDim.x){
            vals[idxS-first] = AS[idxS];
            cols[idxS-first] = JA[idxS];
        }
        __syncthreads();
        //Quarto ciclo for: esegue il calcolo accedendo alla shared memory
        for(int t = tid; t<(last-first)/maxnz[submatIdx]*n; t +=blockDim.x){
            for (int j=0; j<maxnz[submatIdx]; j++){
                res += vals[(t/n)*maxnz[submatIdx] + j] * coeff[cols[(t/n)*maxnz[submatIdx] + j]*n + (t%n)];
            }
            myRes[(hackSize*submatIdx + counter + t/n)*n + (t%n)] = res;
            res=0;
        }
        counter += (last-first)/maxnz[submatIdx];
        __syncthreads();
    }
}
else{
    for(int t = tid; t<hackSize*n; t +=blockDim.x){
        for (int j=0; j<maxnz[submatIdx]; j++){
            res += AS[hackOffsets[submatIdx]+(t/n)*maxnz[submatIdx] + j] * coeff[JA[hackOffsets[submatIdx]+(t/n)*maxnz[submatIdx] + j]*n + (t%n)];
        }
        myRes[(hackSize*submatIdx + t/n)*n + (t%n)] = res;
        res=0;
    }
    __syncthreads();
}
}
```

### 3. Risultati

I risultati sono stati raccolti considerando un set di trenta matrici di test in formato MatrixMarket. In maniera iterativa si è preso in considerazione una matrice per volta, si è proceduto con la sua conversione in formato COO e da qui poi nel formato CSR ed ELLPACK. Dopodichè è stato creato un multivettore per ogni numero di colonne nel set  $\{3, 4, 8, 12, 16, 32, 64\}$  e sono stati eseguiti i seguenti test, ognuno ripetuto per dieci volte:

- calcolo del prodotto seriale
- calcolo del prodotto in CUDA con CSR
- calcolo del prodotto in CUDA con H-ELLPACK al variare dell'hacksize (32 o 64)
- calcolo del prodotto in openMP con CSR al variare del numero dei thread (da 1 a 40).
- calcolo del prodotto in openMP con ELLPACK al variare del numero dei thread (da 1 a 40).

Per ognuna di queste esecuzioni, vengono memorizzati in file csv il tempo di esecuzione, lo speedup, la larghezza di banda, i GFLOPS e l'errore relativo medio tra tutti gli elementi della matrice, ottenuto confrontando il risultato del codice seriale con quello parallelo.

E' stato utilizzato il tipo di dato *double* per la rappresentazione in memoria dei valori decimali.

In particolare, le metriche sopra citate sono state calcolate come segue:

- $Speedup = \frac{\text{tempo in parallelo}}{\text{tempo in seriale}}$
- $GFLOPS = \frac{2 * k * NZ}{T} * 10^{-9}$ , dove k rappresenta il numero di colonne del multivettore, NZ il numero di non zeri della matrice e T il tempo di esecuzione.
- $Banda(GB/s) = \frac{\text{Byte scritti} + \text{byte letti}}{\text{tempo di esecuzione}} * 10^{-9}$
- $Errore relativo = \frac{|y_{\text{seriale}} - y_{\text{parallelo}}|}{|y|} * 10^{15}$

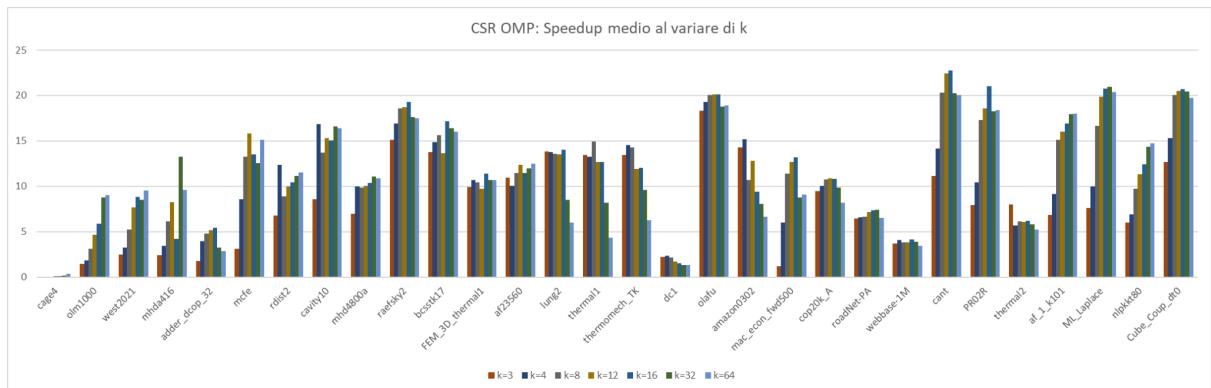
Queste prestazioni sono state ottenute eseguendo su una scheda NVIDIA Quadro - RTX5000 e su un processore Intel Xeon Silver 4210 CPU-2.20Ghz.

### 3.1. Prestazioni prodotto OpenMP

Di seguito vengono riportati dei grafici rappresentanti le prestazioni ottenute dagli algoritmi in OpenMP con i formati CSR ed ELLPACK. Per quanto riguarda gli errori relativi, misurati confrontando i risultati ottenuti in OpenMP con il risultato dell'algoritmo del seriale, sono stati ottenuti valori pari a 0, considerando valori dell'ordine di  $10^{-15}$  e, quindi, non sono riportati in un grafico.

#### 3.1.1. CSR

Il seguente grafico rappresenta lo speedup medio al variare del numero di colonne k del multivettore per tutte le matrici di test.



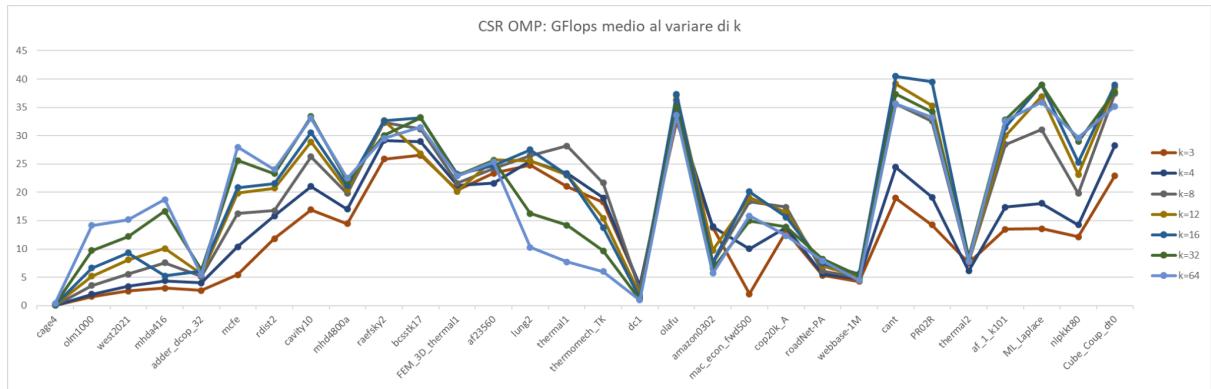
Analizzando l'andamento, si nota che in generale abbiamo prestazioni superiori al calcolo seriale, con qualche eccezione:

- Matrici molto piccole (cage4) hanno prestazioni peggiori poiché l'overhead dovuto alla creazione dei thread è alto rispetto al tempo necessario per calcolare il prodotto. Questo risultato è vero in generale anche nel caso di Ellpack e in prodotti eseguiti in GPU.
- Matrici in cui esistono una o più righe molto grandi rispetto alle altre (es. dc1) sono molto difficili da gestire nel caso di prodotto parallelo poiché un singolo thread si

troverà a dover eseguire molto più lavoro degli altri. Comunque, in questo caso, presentano prestazioni leggermente migliori rispetto al prodotto seriale.

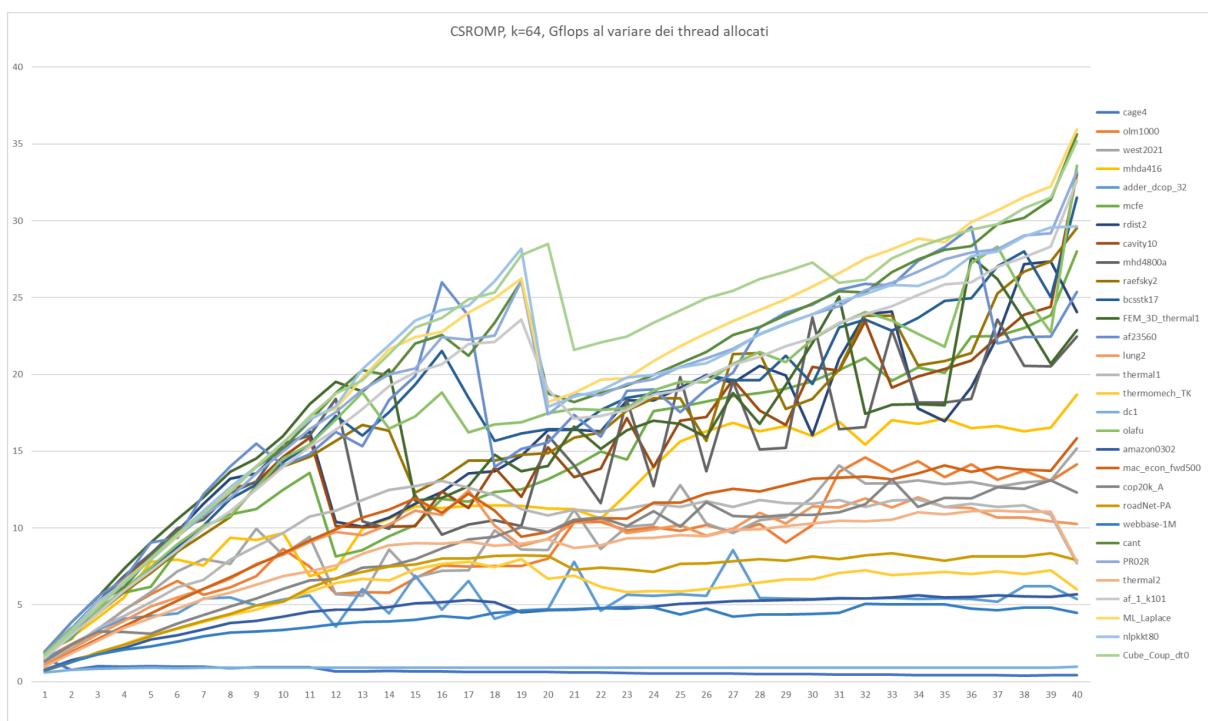
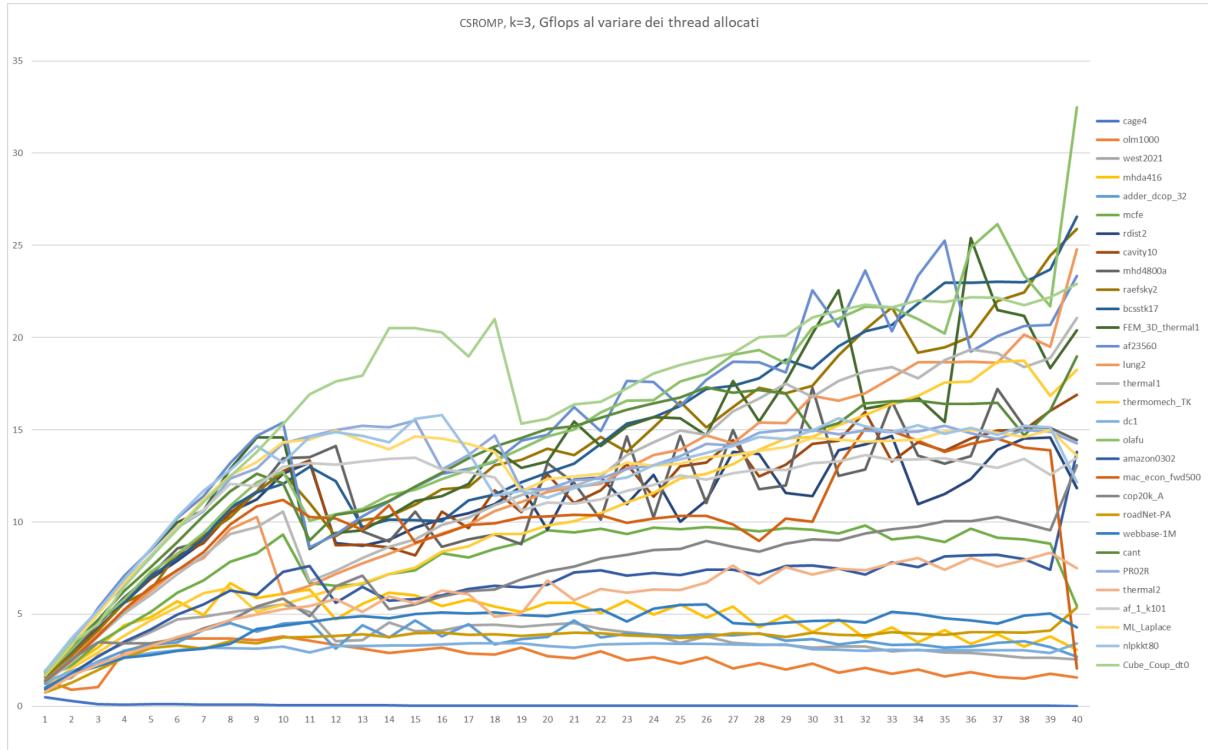
Quasi in tutti i casi si nota come per una dimensione del multivettore maggiore si hanno migliori prestazioni. Questa tendenza verrà, ovviamente, confermata anche dalle altre metriche considerate.

Il seguente grafico mostra l'andamento medio dei GFLOPS al variare del numero di colonne k del multivettore per tutte le matrici di test.



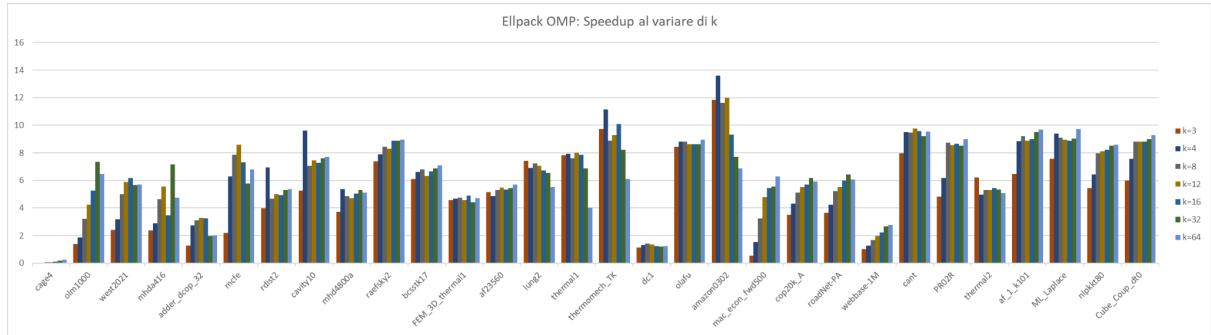
Anche in questo caso possiamo vedere come, in generale, per k bassi abbiamo prestazioni peggiori. Inoltre, considerando le matrici ordinate per numero di non zeri, vediamo come matrici a sinistra tendono ad andare meglio con k=32, mentre le matrici presentano Gflops più alti se moltiplicati con multivettori con k=64. Ci sono però alcune eccezioni. Le prestazioni massime raggiunte dal codice implementato si aggirano intorno ai 40 GFLOPS.

I due grafici che seguono mostrano l'andamento dei GFLOPS al variare del numero di thread (da 1 a 40) utilizzati per il calcolo, fissati k pari a 3 (nel primo) e k pari a 64 (nel secondo). Nell'analisi bisogna tenere conto che il processore utilizzato presenta 10 core per socket, due socket e 2 thread per ogni core, per un totale di 40 thread. Non ha senso aumentare il numero di thread oltre i 40 poiché i tempi di accesso ad un core fisico diventerebbero non trascurabili e peggiorerebbero le prestazioni.

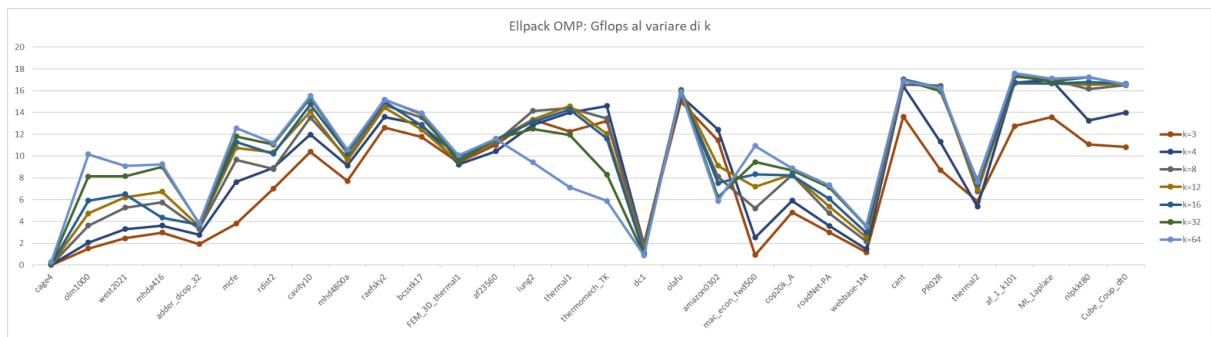


Possiamo notare in generale che, per matrici grandi, otteniamo prestazioni migliori all'aumentare del numero di thread dopo un calo concentrato tra i 10 e i 20 thread. Questo perché la mole di lavoro da fare è maggiore e, quindi, ogni thread contribuisce al calcolo del risultato. Per matrici piccole, invece, aumentando il numero di thread non si ottengono miglioramenti poiché le computazioni da eseguire sono poche e i thread creati vengono in qualche modo "sprecati" perché non hanno abbastanza lavoro da fare da giustificare la creazione.

### 3.1.2. Ellpack

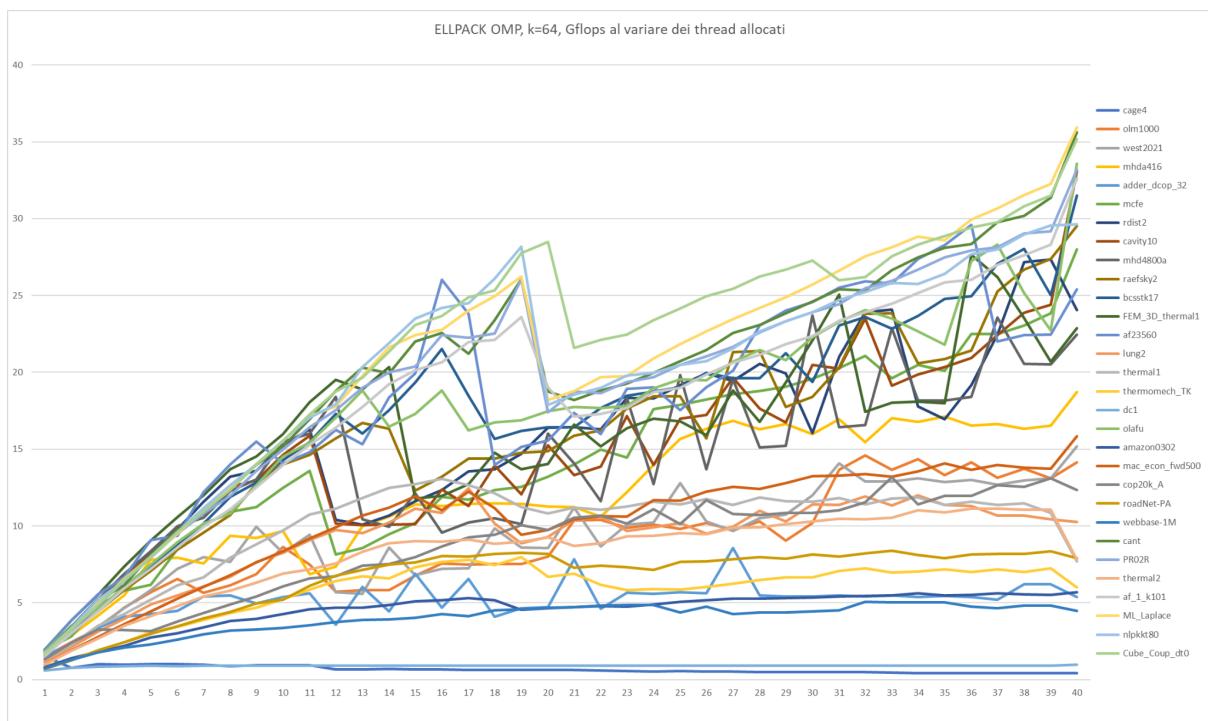
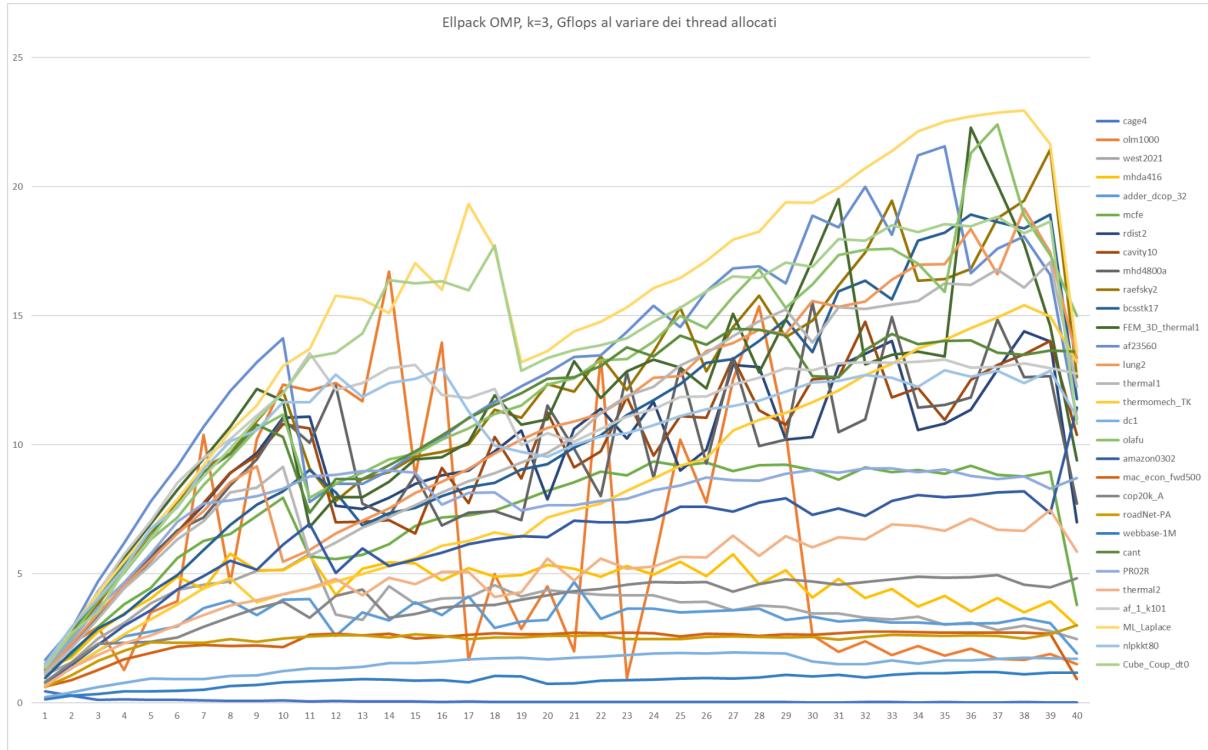


Analizzando lo speedup nel caso di Ellpack, in generale otteniamo le stesse considerazioni discusse nel caso di CSR per le matrici piccole e le matrici con righe molto grandi. Inoltre, si nota che CSR ha speedup più alto rispetto a Ellpack.



Le informazioni ottenute dallo studio sullo speedup sono confermate analizzando il grafico dell'andamento dei GFLOPS al variare del numero di colonne del multivettore per tutte le matrici. Vediamo come le prestazioni massime raggiunte si aggirano intorno ai 18 GFLOPS.

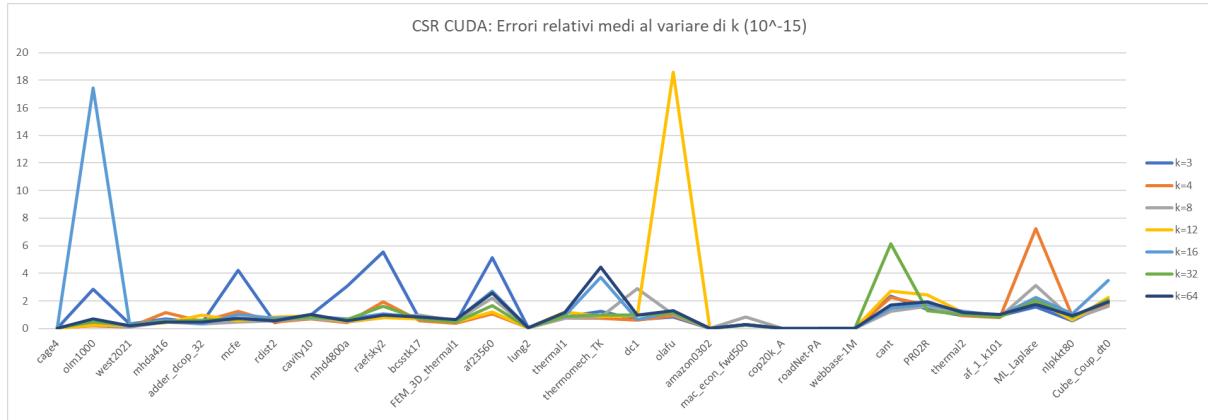
Di seguito troviamo due grafici che mostrano l'andamento dei GFLOPS al variare del numero di thread per k=3 e k=64.



Al variare dei thread allocati abbiamo un comportamento simile a quello ottenuto in CSR, ma si può notare come per  $k=3$  ci sia un decadimento delle prestazioni nell'allocazione di 40 thread.

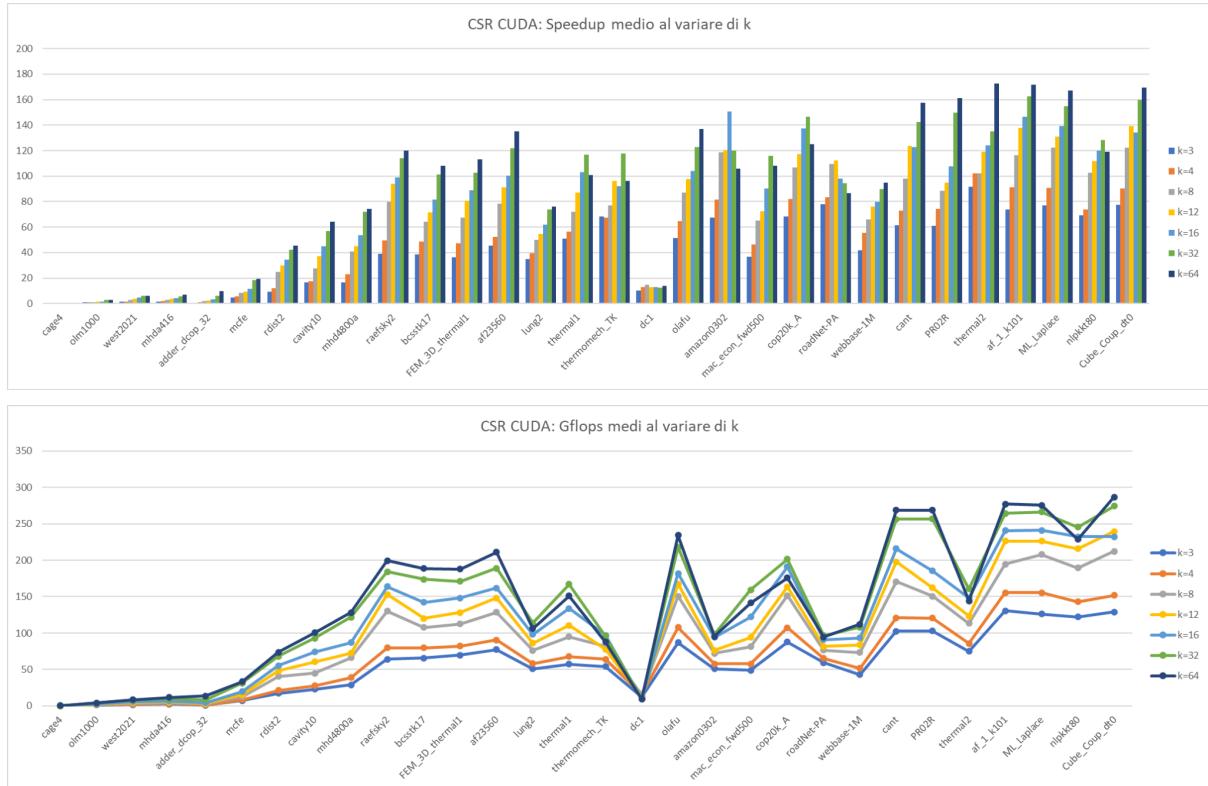
## 3.2. Prestazioni prodotto CUDA

### 3.2.1. CSR



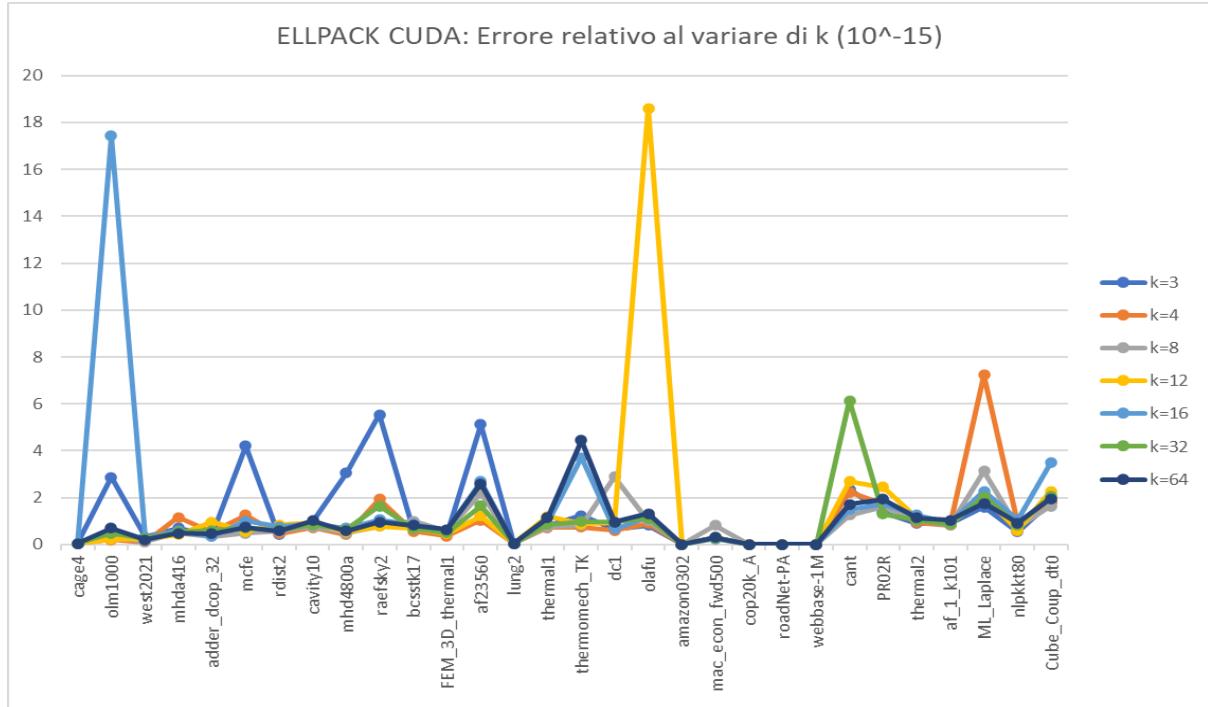
Analizzando il grafico riguardante l'errore relativo, possiamo notare come in generale, considerando come tipo di dato il *double*, si ottengono errori dell'ordine di  $10^{-15}$ , tranne in casi particolari in cui abbiamo picchi dell'ordine di  $10^{-14}$ .

Di seguito sono riportati i grafici rappresentanti l'andamento dello speedup e dei GFLOPS al variare del numero di colonne del multivettore per tutte le matrici.



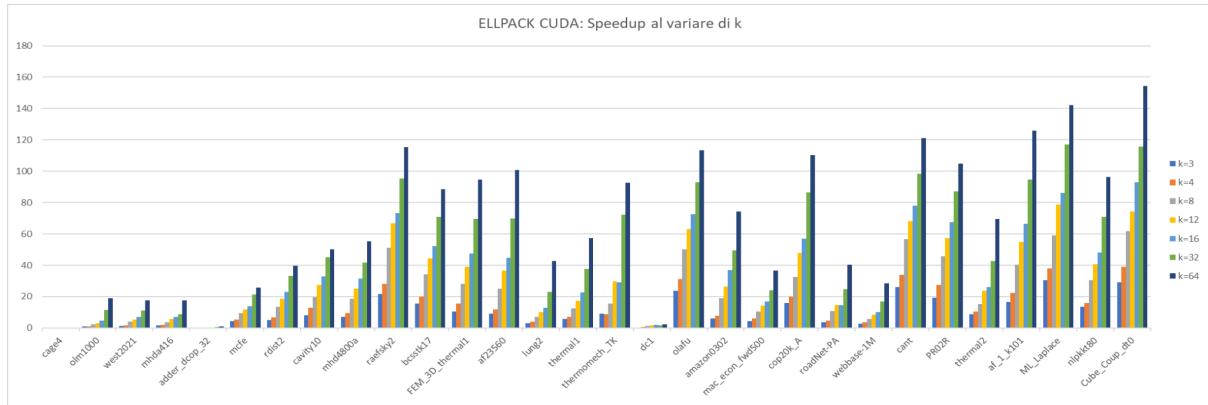
Gli andamenti presentano le stesse caratteristiche di quelli presentati per OpenMP ma, vista l'elevata potenza di calcolo utilizzabile della GPU, si assestano su valori molto maggiori. In particolare, infatti, con 64 colonne si sfiorano i 300 GFLOPS e ci si avvicina abbastanza, quindi, al massimo teorico fissato a 330 GFLOPS.

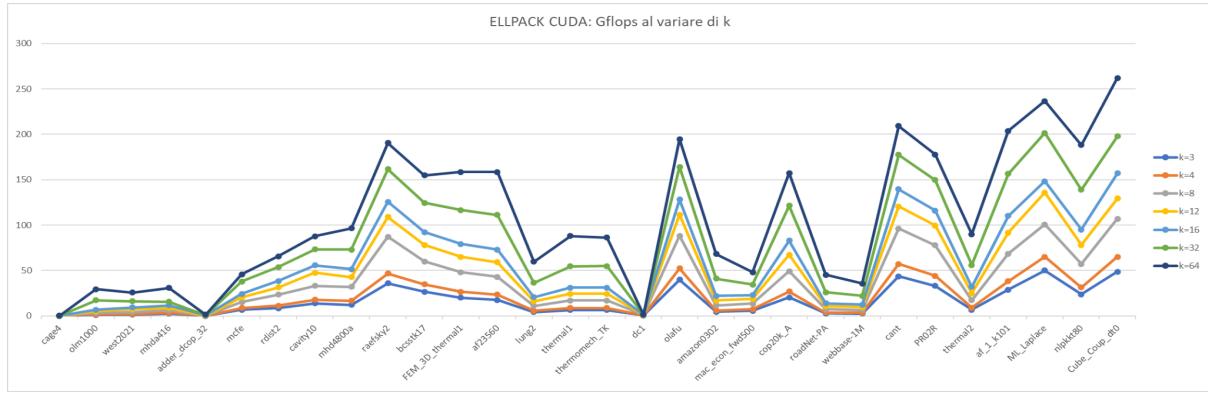
### 3.2.2. Ellpack



Per quanto riguarda l'errore relativo in Ellpack GPU, notiamo che gli errori sono quasi identici a quelli ottenuti nel prodotto in formato CSR. Questo ci mostra come l'errore relativo ottenuto sia fortemente dipendente dalla matrice e non dall'implementazione del prodotto.

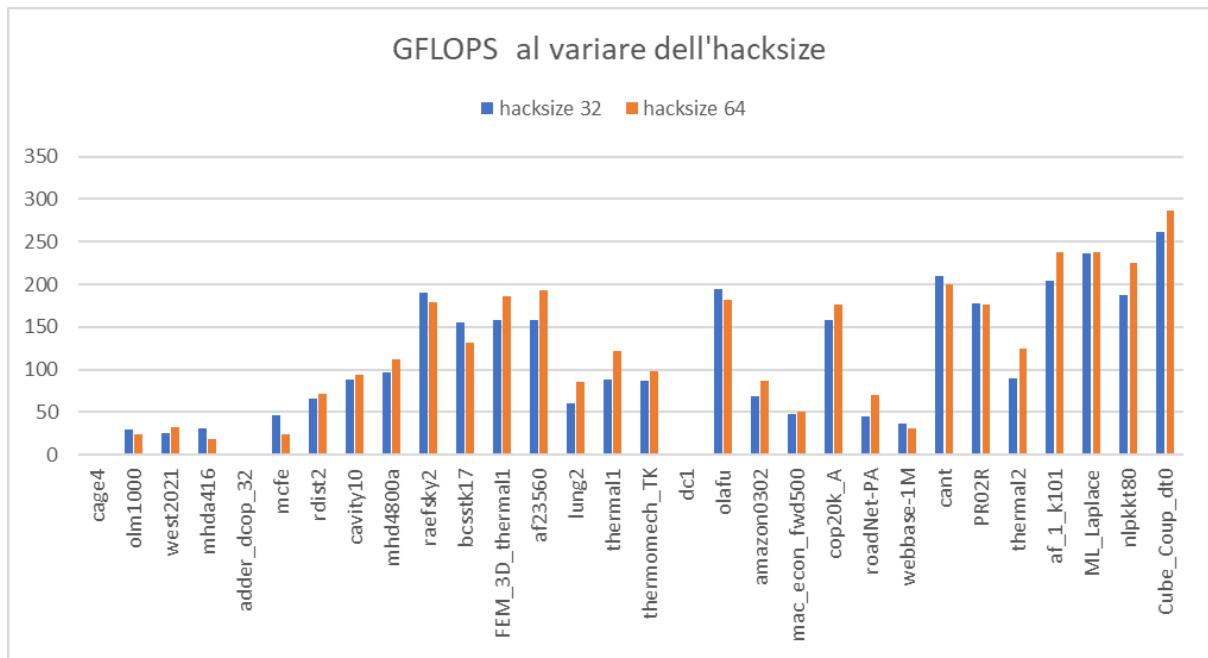
I grafici seguenti mostrano l'andamento dello speedup e dei GFLOPS al variare del numero di colonne e delle matrici, fissato un hackszie pari a 32.

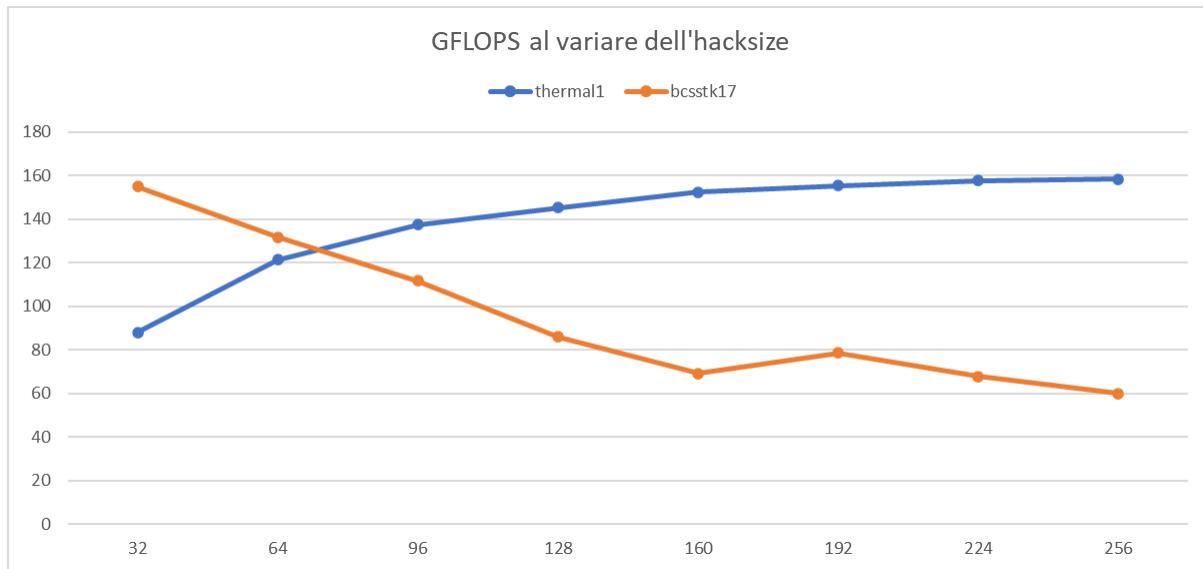




Anche questi grafici di speedup e Gflops confermano quanto detto in precedenza. In Ellpack notiamo che, all'aumentare di  $k$ , aumentano in generale le prestazioni, che raggiungono un massimo di poco più di 250 GFLOPS.

Di seguito sono riportati due grafici che mostrano l'andamento dei GFLOPS al variare dell'hacksize e al variare delle matrici. In particolare, nel primo grafico sono stati considerati valori dell'hacksize solamente pari a 32 e 64 poiché, per valori più elevati, alcune matrici non permettevano di allocare lo spazio necessario sul dispositivo. Nel secondo grafico, quindi, sono state selezionate due matrici e si è fatto variare l'hacksize da 32 a 256 con passi di 32.



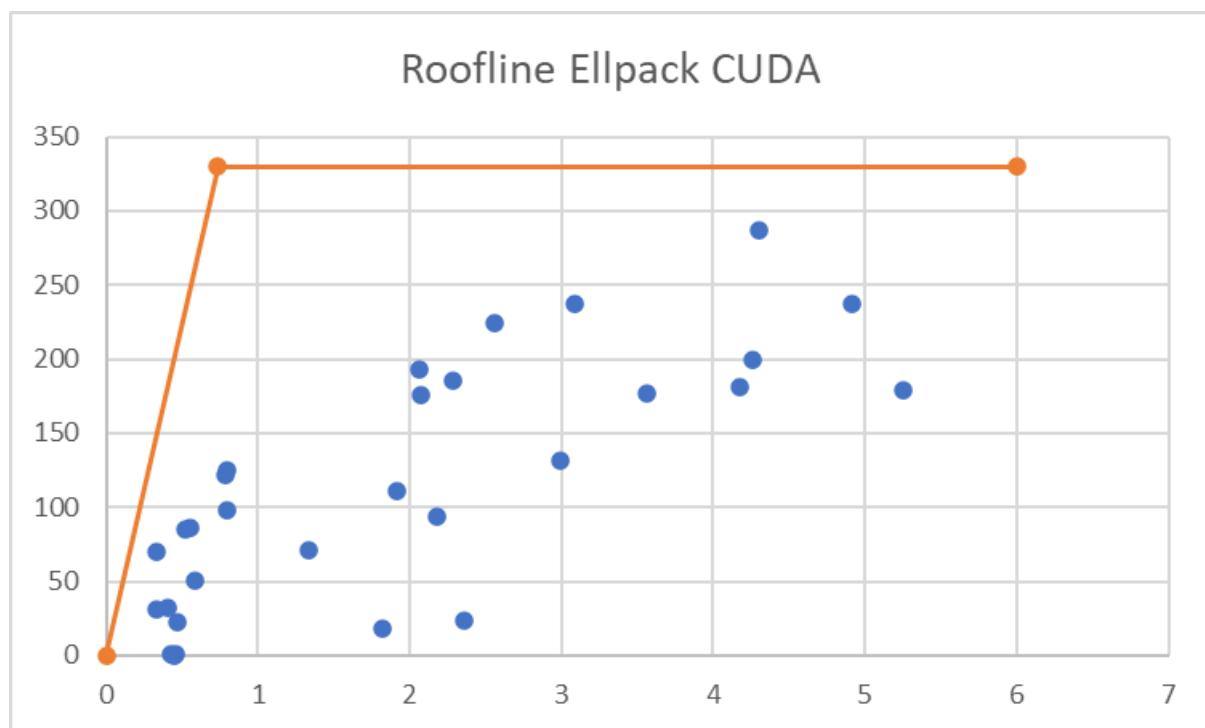
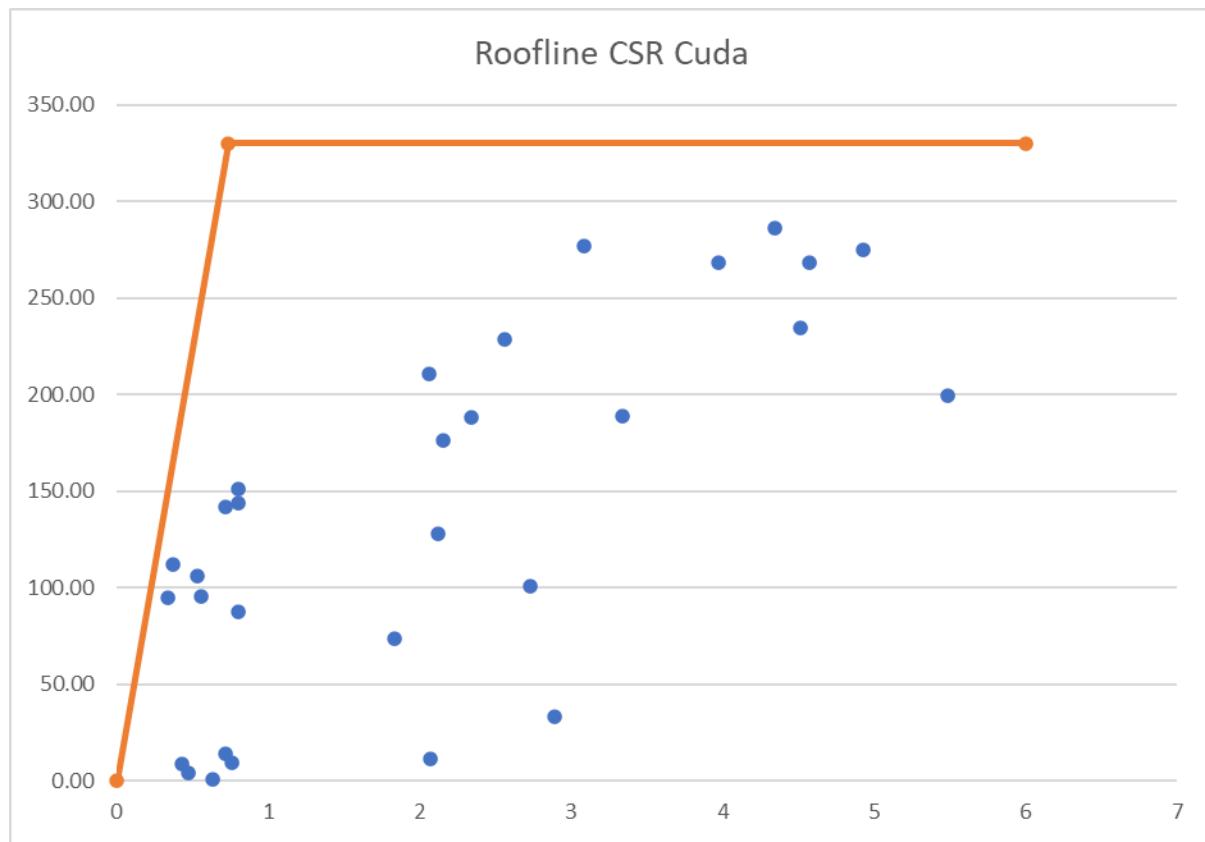


Come possiamo vedere dal primo grafico, in generale un aumento dell'hacksize porta a migliori prestazioni. Questo però, come si evince anche dal secondo grafico, non è vero per tutte le matrici. Il motivo potrebbe essere legato strettamente all'implementazione: per sfruttare quanto più possibile la shared memory, nel caso in cui la sottomatrice diventi troppo grande essa viene suddivisa ulteriormente in sotto-sotto matrici nel blocco, gestite una alla volta dai thread del blocco stesso. Aumentando l'hacksize, quindi, la probabilità che questo possa accadere è maggiore.

### 3.2.3 Modello Roofline

Questo modello è stato utilizzato per avere una visualizzazione grafica delle prestazioni ottenute in funzione della banda di memoria e della potenza di calcolo disponibili sulla scheda grafica e in funzione dell'intensità aritmetica. Quest'ultima è definita come  $\frac{GFLOPS}{Bandwidth}$ . L'upper bound del grafico rappresenta il limite massimo teorico delle prestazioni ottenibile ed è definito come  $\min(\max GFLOPS, AI * \max bandwidth)$  dove  $\max GFLOPS$  sta per il limite teorico massimo dei gflops fissato a 330 GFLOPS, AI sta per l'intensità aritmetica e  $\max bandwidth$  sta per il limite teorico massimo della banda pari a 448 GB/s.

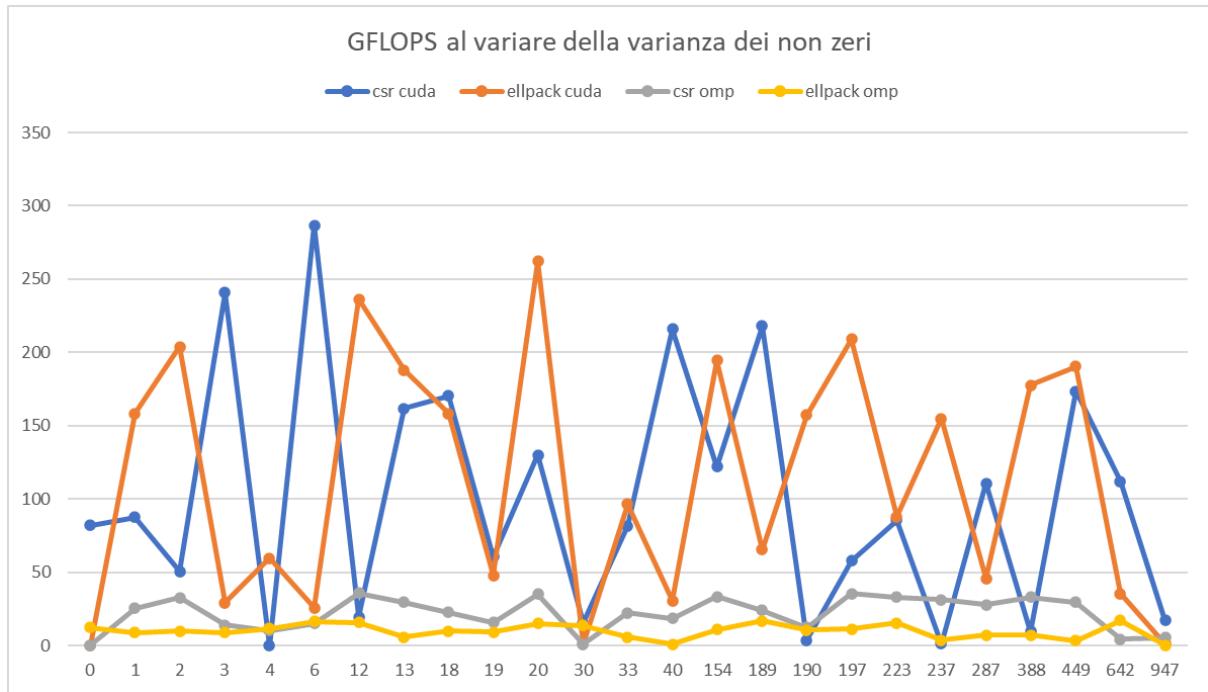
I grafici seguenti, quindi, mostrano l'andamento dei GFLOPS al variare dell'intensità aritmetica calcolata per tutte le matrici. Come si può vedere sia nel grafico relativo a CSR sia nel grafico relativo ad ELLPACK, le prestazioni di alcune matrici, rappresentate dai punti, si avvicinano abbastanza ai limiti massimi teorici, suggerendo buoni livelli di parallelizzazione ed utilizzo della memoria.



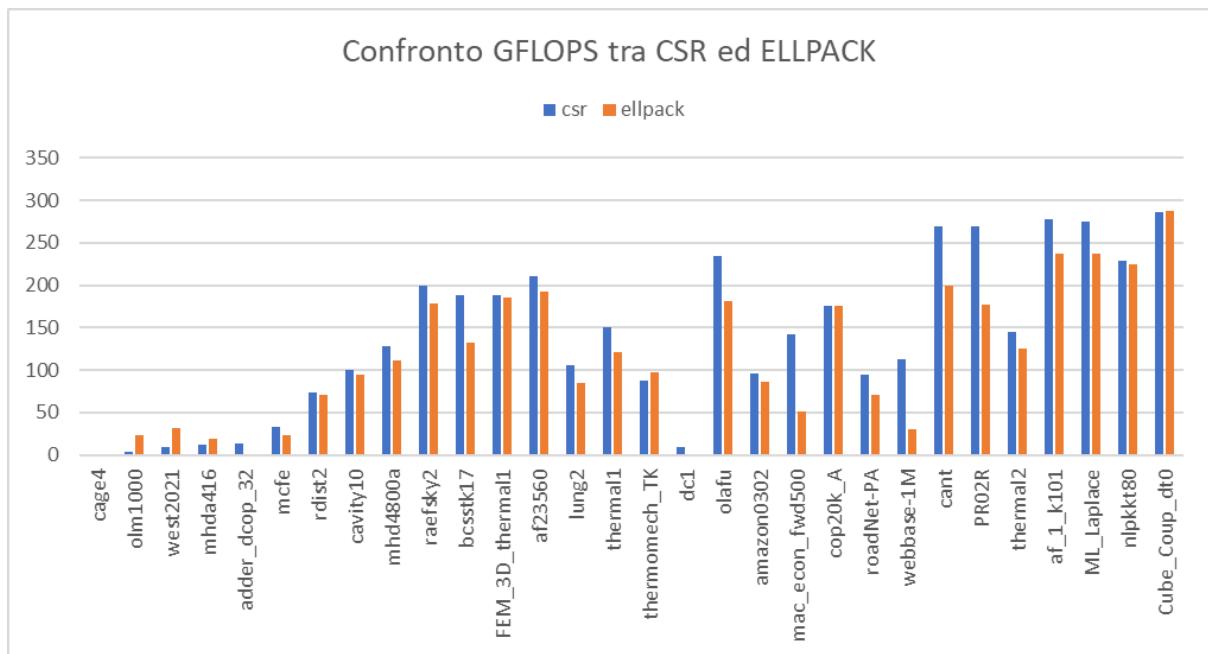
### 3.3 Considerazioni generali

Il grafico seguente mostra l'andamento dei GFLOPS al variare della varianza dei non zeri tra le righe delle matrici. Come si può vedere dall'andamento altalenante per tutti gli algoritmi di

calcolo, la varianza dei non zeri non influisce in modo significativo sulle prestazioni calcolate in GFLOPS.



Il grafico seguente mostra un confronto delle prestazioni tra CSR ed ELLPACK in CUDA. Per Ellpack è stato fissato l'hackszie a 64. Come si può notare, in generale l'implementazione fornita in CSR sembra avere prestazioni migliori, tranne che per alcune matrici in cui le prestazioni sono pressoché identiche e per le matrici più piccole in cui ELLPACK presenta prestazioni migliori.



## 4. Conclusioni

In sintesi, possiamo dire che, sia nel caso di prodotto in GPU, sia in quello in CPU, l'algoritmo che sfrutta il formato CSR ha prestazioni migliori rispetto a quello che utilizza il formato Ellpack. Possiamo però notare dal modello Roofline che abbiamo ancora margini di miglioramento, soprattutto per determinate tipologie di matrici. In generale, abbiamo visto come la dimensione delle matrici, oltre alle dimensioni troppo elevate di una singola riga di una matrice, siano fattori determinanti per le prestazioni degli algoritmi proposti. Inoltre, nel caso di matrici Ellpack, sono altresì importanti il padding e, nel caso di H-Ellpack, la dimensione in righe delle sottomatrici (che di fatto determina una variazione del padding). Per quanto riguarda il multivettore, i risultati ci mostrano come, in generale, l'aumento del numero di colonne porti a un aumento delle prestazioni, con qualche eccezione in cui si ottengono le prestazioni migliori per k=32.

## 5. Suddivisione del lavoro

Tutte le attività e organizzazione del codice, le analisi delle possibili ottimizzazioni, la generazione dei grafici e l'analisi dei dati sono state effettuate in coppia.

<b>Federica Villani</b>	<b>Mattia Antonangeli</b>
Implementazione formato CSR	Implementazione formato Ellpack
Conversione matrix market - coo	Implementazione formato H-ELLPACK
Prodotto seriale CSR	Generazione multivettore
Prodotto OpenMP CSR	Prodotto OpenMP Ellpack
Prodotto CUDA CSR-Adaptive	Prodotto CUDA Ellpack
Prodotto CUDA H-ELLPACK	