Progettazione ed analisi delle performance di algoritmi per il calcolo del SpMV su CPU e GPU

Pasquale Caporaso

Universita degli di studi di Roma - Tor Vergata

1 Introduzione

Il prodotto di una matrice sparsa per un vettore denso è un'operazione estremamente comune in ambito scientifico e la complessità sempre maggiore dei modelli che necessitano questo tipo di calcolo rende necessaria l'ideazione di algoritmi sempre più efficenti per completare questa operazione. In questa relazione viene descritto come questo problema viene affrontato e le performance ottenute su due diverse architetture di calcolo, la GPU e la CPU.

2 Descrizione del problema

Il problema del SpMV è all'apparenza un problema semplice, la motiplicazione di una matrice per un vettore, tuttavia la matrice ha una proprietà particolare, è una matrice "sparsa", ovvero una matrice in cui quasi tutti i valori sono uguali a zero o, per dare una definizione più formale:

$$NZ << N * M$$

dove NZ è il numero degli elementi diversi da 0, N è il numero di righe e M il numero di colonne della matrice. Il prodotto matrice-vettore e già, per natura, un'operazione pesante, in quanto esegue $O(N^2)$ operazioni in virgola mobile per $O(N^2)$ elementi, questo porta ad un alto rapporto tra operazioni e accessi a memoria, ad uno scarso riuso di elementi e quindi ad una scarsa località temporale. Chiaramente, aggiungendo a tutto questo il fatto che la matrice è quasi completamente "vuota" la situazione non può che peggiorare.

Un altro fattore che aggiunge alla complessità di questo problema è che il salvataggio di una matrice sparsa non avviene in maniera convenzionale, ovvero un array bidimensionale, ma bensi con dei formati alternativi, studiati per mantenere informazioni solo sui non-zeri all'interno della matrice, chiaramente il formato di salvataggio ha un impatto sostanziale sulle performance dell'algoritmo e, nel corso degli anni ne sono stati inventati diversi, ognugno con i sui pregi e con i suoi difetti. In questa relazione userò solamente a due di questi, tra i più popolari, il formato ELLPACK e il formato CSR.

3 Lettura delle matrici

Le matrici esaminate in questo studio sono state quelle suggerite, ovvero le seguenti:

cage4	mhda416	mcfe
olm1000	adder dcop 32	west2021
cavity10	rdist2	cant
olafu	Cube Coup dt0	ML Laplace
bcsstk17	mac econ fwd500	mhd4800a
cop20k A	raefsky2	af23560
lung2	PR02R	FEM 3D thermal1
thermal1	thermal2	thermomech TK
nlpkkt80	webbase-1M	dc1
amazon0302	af 1 k101	roadNet-PA

Queste matrici sono state scaricate dalla SuiteSparse Matrix collection ed erano tutte salvate nel formato MatrixMarket ed ordinate per colonna invece che per riga.

Per questo motivo è stato necessario salvare i valori del file in delle liste ordinate per riga come passo intermedio prima di poter effettivamente organizzare i dati nei due formati CSR ed ELLPACK. In questo modo il tempo di lettura rimane lineare nel numero di non-zeri, tuttavia questa operazione rimane più lenta dell'effettivo prodotto per la sua interazione con il file-system.

Tutte le funzioni che si occupano della lettura delle matrici e le strutture che rappresentano i due formati finali si trovano nei file "formats.h" e "formats.c". Questa libreria utilizza la "mmio.c" messa a disposizione dagli sviluppatori dello standard MatrixMarket ed espone delle funzioni che permettono di ottenere matrici in formato CSR e ELLPACK passando semplicemente il path del file. La libreria gestisce anche matrici simmetriche e quelle salvate come pattern ricostruendo il triangolo mancante nel caso delle prime e inserendo come valore dell'elemento 1 nel caso delle seconde.

Infine, faccio notare che le matrici **ja** ed **as** nel formato ELLPACK sono salvate in realtà come vettori ed gli indici corretti per l'accesso ai dati sono calcolati manualmente ogni volta, ho fatto questa scelta di design perchè il linguaggio C non permette l'utilizzo di matrici con grandezza definita a runtime.

4 Implementazione degli algoritmi

4.1 CPU - Seriale

L'algoritmo seriale è servito come base di partenza per misurare la velocità di quelli paralleli e anche per assicurare che il valore finale del vettore risultato fosse corretto. Ho usato l'implementazione standard per i due formati, con qualche leggera accortezza per non ripetere letture in memoria dove non necessario, il codice, in linguaggio C, è il seguente:

```
1  //csr
2  unsigned int i,j,limit;
3  double sum;
4  for (i = 0; i < M; i++) {
5          sum = 0;
```

```
        limit = irp[i+1];
        for (j = irp[i]; j < limit; j++)
                sum += as[j]*array[ja[j]];

        result[i] = sum;
}
```

```
//ellpack
unsigned int i,j,row;
double sum;
for (i = 0; i < M; i++) {
        sum = 0;
        row = i*maxnz;
        for (j = 0; j < maxnz; j++)
                sum += as[row+j]*array[ja[row+j]];

        result[i] = sum;
}
```

4.2 CPU - Parallelo

Il codice parallelo per CPU è molto simile a quello seriale con la semplice aggiunta di una direttiva Openmp, distribuire i thread sulle righe è stata una scelta naturale, questo ha infatti eliminato eventuali dipendenze tra un thread e l'altro. Per quanto riguarda il tipo di scheduling dei thread il tipo static ha riportato prestazioni migliori rispetto al dynamic, indipendentemente dal chunk_size, questo probabilemente è dovuto al fatto che il numero di operazioni per ciascuna riga è relativamente simile. Il codice C della implementazione parallela è il seguente:

```
//csr
unsigned int i, j, limit;
double sum;
#pragma omp parallel for schedule(static) default(none)  \\
    private(i, j, sum, limit) shared(M, ja, irp, as, array, result)
for (i = 0; i < M; i++) {
    sum = 0;
    limit = irp[i+1];
        for (j = irp[i]; j < limit; j++)
                sum += as[j]*array[ja[j]];

        result[i] = sum;
}
```

```
//ellpack
unsigned int i, j, row;
double sum;
```

```
#pragma omp parallel for schedule(static) default(none) \\
    private(i, j, sum, row) shared(matrix, M, ja, as, maxnz, array, result)
for (i = 0; i < M; i++) {
        sum = 0;
        row = i*maxnz;
        for (j = 0; j < maxnz; j++)
                sum += as[row+j]*array[ja[row+j]];

        result[i] = sum;
}
```

5 GPU

Per quanto riguarda l'implementazione in CUDA ho deciso di assegnare la moltiplicazione di una riga ad un singolo thread in quanto il numero di elementi per riga è relativamente piccolo, ho inoltre organizzato questi thread in un grid-stride loop per mantenere flessibile il numero di blocchi e per ottenere la massima memory coalescence. Alla fine il numero ottimo di blocchi si è rivelato essere pari al numero multiprocessori con 128 thread per blocco, aumentare questi due valori causava un peggioramento delle prestazioni, questo significa che queste impostazioni raggiungono la massima larghezza di banda disponibile. Il codice C dell'implementazione è il seguente:

```
//csr
int tid = threadIdx.x + blockDim.x * blockIdx.x;
int i, j;

//grid-stride
for(i = tid; i < M; i += blockDim.x*gridDim.x) {

  // thread sums row
  int limit = irp[i+1];
  int start = irp[i];
  double sum = 0;
  for (j = start; j < limit; j += 1) {
     sum += as[j] * array[ja[j]];
  }

  result[i] = sum;
}
```

```
//ellpack
int tid = threadIdx.x + blockDim.x * blockIdx.x;
int i, j;

//grid-stride
for(i = tid; i < M; i += blockDim.x*gridDim.x) {
```

```
7
8       // thread sums row
9       double sum = 0;
10      int index = i*maxnz;
11      for (j = 0; j < maxnz; j += 1) {
12          sum += as[index+j] * array[ja[index+j]];
13      }
14
15      result[i] = sum;
16  }
```

6 Analisi dei risultati

6.1 Numero ottimale di thread per la CPU

Prima di procedere con il confronto tra architetture diverse bisogna discutere qual è il numero ottimale di thread in CPU da usare per ottenere le performance migliori, questo non è banale, dato che questo calcolo è fortemente memory bound potremmo raggiungere un punto in cui abbiamo raggiunto il massimo della banda della memoria avendo avendo un numero di thread inferiore al numero di core, logici o fisici. Questo è effettivamente quello che succede, come mostrato in figura 1 e 2, il prodotto della maggior parte delle matrici ha un forte calo di prestazione quando si raggiunge il massimo dei core fisici, ovvero 20, ritengo che sia questo quindi il numero ottimale di thread da usare e sarà questo il valore riportato nelle prossime sezioni quando comparerò la performance su CPU a quella di GPU.

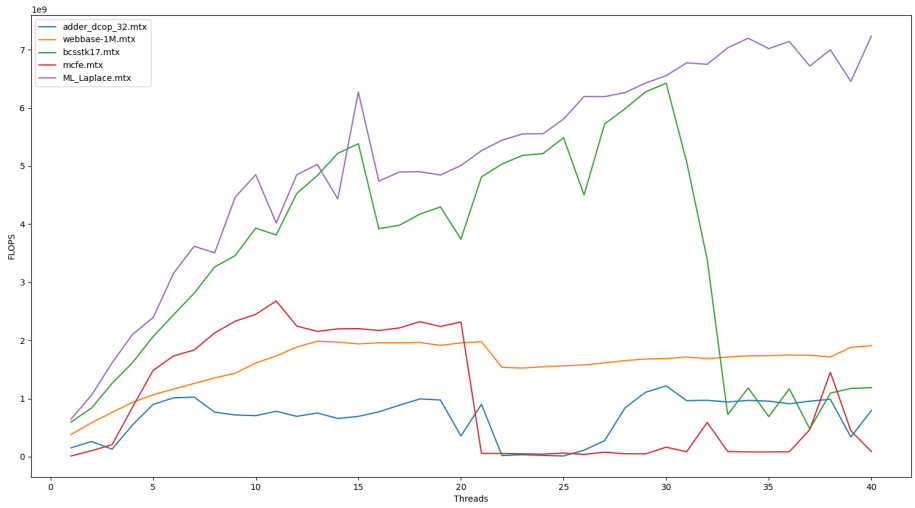

Figure 1: Performance su CPU - formato CSR

Per quanto riguarda la differenza di performance tra le matrici stesse, questo è naturale, il programma ha prestazioni migliori quando le matrici sono più grandi in quanto il numero di non-zeri è diviso tra molteplici thread.

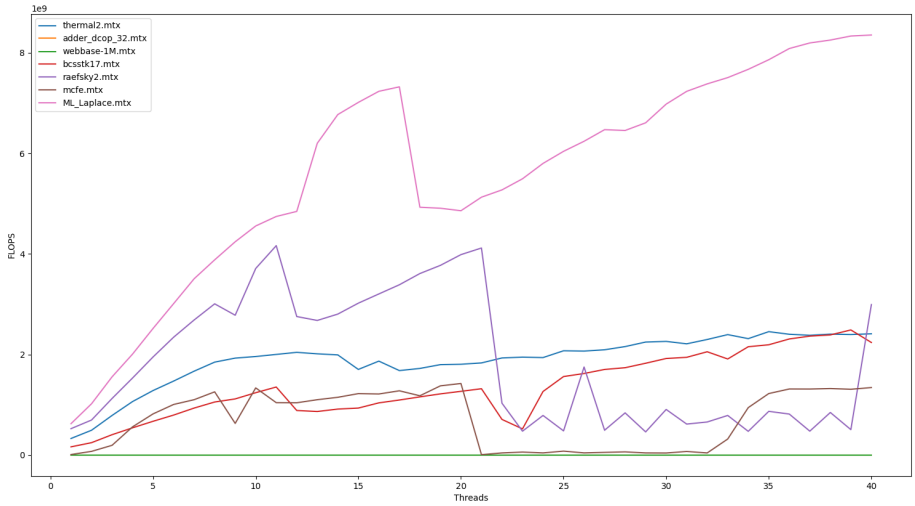

Figure 2: Performance su CPU - formato ELLPACK

6.2 Performance GPU e confronto con CPU

La GPU ha naturalmente una performance molto maggiore in questo tipo di problemi e questo si può osservare in figura 3 e 4, vediamo che la maggioranze delle matrici vengono moltiplicate più velocemente in GPU, con uno speed-up massimo di 15/16 e una media di circa 6. Le uniche eccezioni sono matrici piccole come la "cage4" che contiene solo 9 righe, in situazioni come questa la potenza computazionale della GPU non riesce a mascherare la latenza della memoria.

Chiaramente però, semplicemente guardare il rapporto con la CPU non garantisce la qualità del codice su GPU, quindi guardiamo i reali valori di performance in tabella 1, come possiamo notare il prodotto viene eseguito ad una veloccità che va da i 25 ai 50 GIGAFLOPS, questa disparità è sempre dovuta alle dimensioni della matrice e al numero di nonzeri. Queste prestazioni normalmente non sarebberò soddisfacenti in quanto la potenza di calcolo di una GPU è sicuramente superiore, tuttavia il SpMV è un calcolo fortemente Memory-Bound, per questo motivo non dobbiamo guardare la potenza di calcolo generale ma bensì la larghezza di banda, in particolare ci si può aspettare una velocità pari ad un sesto della banda. Tutti questi calcoli sono stati effettuati su una Nvidia Quadro RTX 5000 che ha una larghezza di banda di picco di 448 GB/s, ciò significa che la massima velocità raggiungibile è di circa 74 GIGAFLOPS. Bisogna anche considerare che questa è la velocità di picco, non sono riuscito a trovare nessun informazione sulla larghezza di banda sostenuta ma considerata la banda massima credo si possa concludere che 50 GIGAFLOP è una velocità ragionevole per questo tipo di problema su questo particolare scheda.

7 Confronto tra ELLPACK e CSR

Infine, completiamo l'analisi dei risultati osservando le prestazioni dei due formati di salvataggio, ELLPACK e CSR, nella figura 5 ho riportato lo speed up calcolato come ELLPACK/CSR, l'immagine sembrerebbe suggerire che il formato csr ha ripor-

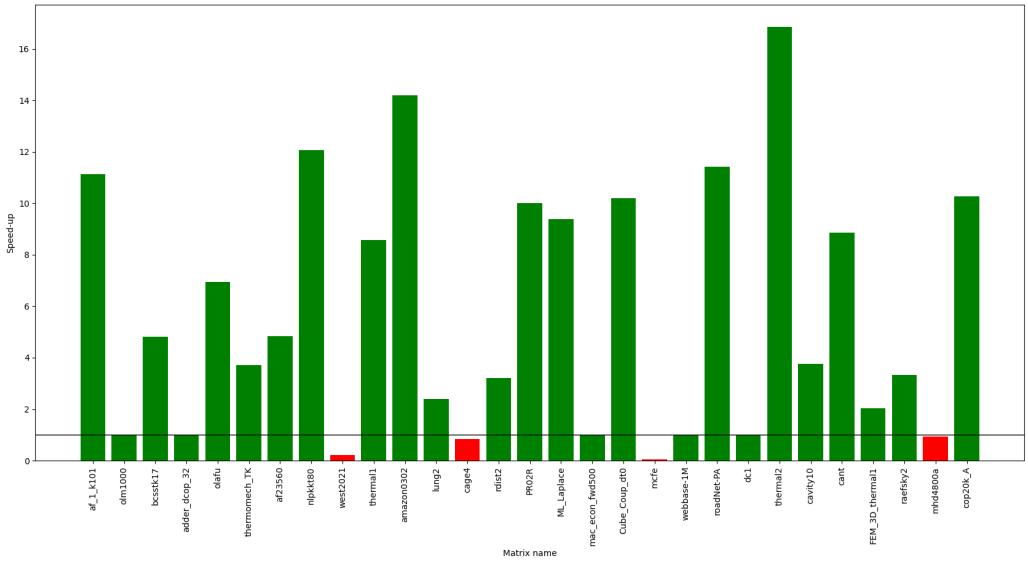

Figure 3: Speed-up ellpack CPU/GPU

tato prestazioni generalmente migliori, tuttavia questo non è necessariamente vero, i risultati vanno analizzati più attentamente.

Il formato ELLPACK ha, infatti, una particolarità, la sua efficenza è dipendente da quanto è "bilanciata" la matrice, questo è dovuto al fatto che i dati sono salvati in una matrice che ha come una delle dimensioni il MASSIMO numero di non-zeri contenuti in un una riga. Ciò vuol dire che se abbiamo una riga con un numero di non-zeri fortemente superiore alle altre il formato ELLPACK diventa inefficente dato che le dimensioni della matrice di salvataggio diventano molto superiori all'effettivo numero di non-zeri. Per quanto le matrici date ho deciso di calcolare la velocità per tutte le matrici in cui il rapporto tra dimensione dati e numero di non-zeri è pari o inferiore a 6, ciò vuol dire che la matrice ELLPACK può contenere fino a 5/6 di valori inutili.

Ora analizziamo come il rapporto tra dimensione matrice e non-zeri ha influenzato l'efficenza del prodotto, possiamo osservare le misurazioni in tabella 2, quello che vediamo è che le prestazioni sono fortemente influenzate la "bilanciamento" della matrice, più il rapporto tra dimensioni del salvataggio e non-zeri è basso più aumenta lo Speed-up con il CSR, il limite dopo il quale il formato CSR diventa più efficente sembra essere intorno a 1.5, mentre per matrici particolamente bilanciate arriviamo ad uno speed-up di circa 1.6. Questo ci mostra che, se trattiamo casi generali, il formato CSR potrebbe essere una scelta migliore mentre, se abbiamo delle matrici in cui il numero di non-zeri è circa uguale per ogni riga il formato ELLPACK performa decisamente meglio.

8 Conclusioni

Il calcolo del SpMV rimarrà sempre una delle operazioni più importanti in ambito matematico ed in questa relazione si è mostrato che il suo calcolo su GPU porta a chiari vantaggi. Per quanto riguarda il formato da utilizzare per salvare una matrice sparsa, questo rimane un problema di fondamentale importanza visto il forte impatto

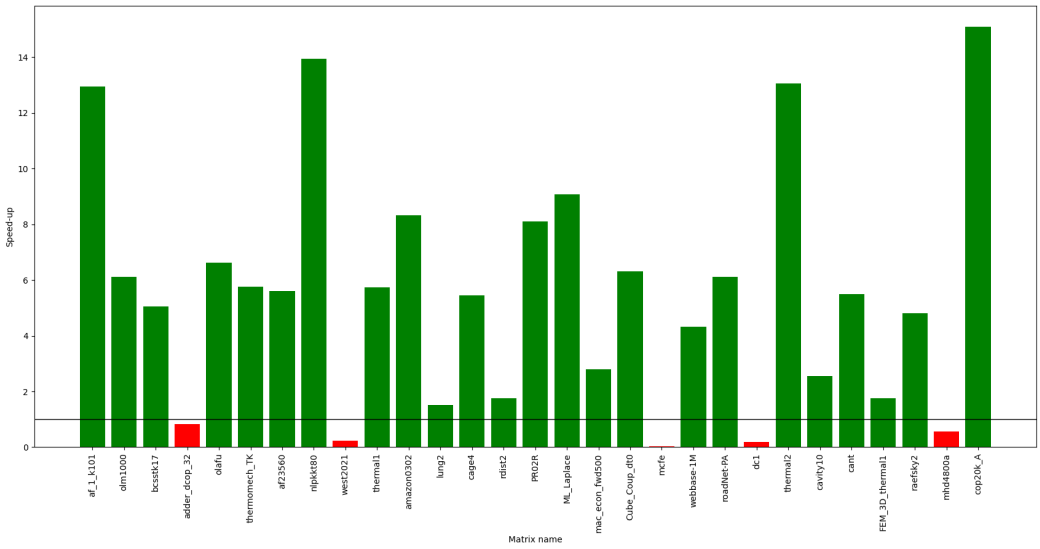

Figure 4: Speed-up csr CPU/GPU

sulle prestazioni, tra i due formati studiati il migliore è il formato ELLPACK a patto che la matrice abbia circa lo stesso numero di non-zeri in ogni riga, se ciò non è vero il formato CSR diventa più efficace.

	CPU		GPU	
Matrix	csr	ellpack	csr	ellpack
af_1_k101	3.98	4.83	51.54	53.78
olm1000	0.15	0.78	0.90	0.79
bcsstk17	3.74	1.27	18.89	6.08
adder_dcop_32	0.36	na	0.30	na
olafu	4.52	3.42	29.98	23.75
thermomech_TK	3.53	2.64	20.29	9.81
af23560	3.86	4.50	21.62	21.80
nlpkkt80	3.63	4.29	50.61	51.76
west2021	0.88	0.80	0.21	0.18
thermal1	4.11	2.85	23.57	24.45
amazon0302	2.84	2.21	23.66	31.31
lung2	3.90	2.51	5.91	6.03
cage4	0.01	0.01	0.01	0.01
rdist2	3.24	1.22	5.68	3.92
PR02R	4.81	2.27	38.97	22.73
ML_Laplace	5.01	4.86	45.47	45.55
mac_econ_fwd500	3.72	na	10.41	na
Cube_Coup_dt0	6.66	3.54	42.02	36.10
mcfe	2.32	1.42	0.08	0.07
webbase-1M	1.96	na	8.47	na
roadNet-PA	2.72	1.35	16.59	15.37
dc1	1.34	na	0.24	na
thermal2	2.13	1.80	27.80	30.43
cavity10	3.88	2.25	9.87	8.43
cant	5.34	3.99	29.30	35.30
FEM_3D_thermal1	4.93	4.22	8.65	8.64
raefsky2	4.65	3.99	22.35	13.27
mhd4800a	4.32	2.74	2.48	2.56
cop20k_A	2.49	1.19	37.52	12.24

Table 1: Performance generali in GIGAFLOPS

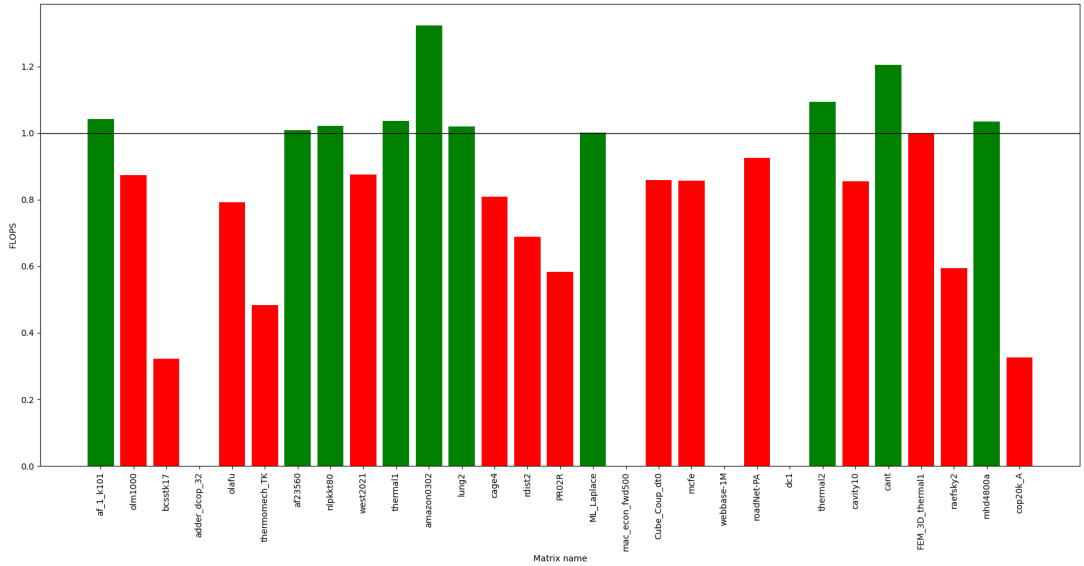

Figure 5: Speed up ELLPACK/CSR

Matrice	Speed-up	Rapporto dati/NZ
af_1_k101	1.0434	1.3948
olm1000	0.8739	1.5015
bcsstk17	0.3219	5.3918
adder_dcop_32	na	211.1889
olafu	0.7923	1.8787
thermomech_TK	0.4836	2.2598
af23560	1.0086	1.0217
nlpkkt80	1.0228	1.9987
west2021	0.8756	3.2982
thermal1	1.0374	2.2641
amazon0302	1.3234	1.0613
lung2	1.0206	1.7778
cage4	0.8086	1.1020
rdist2	0.6896	3.4264
PR02R	0.5832	1.8104
ML_Laplace	1.0020	1.0075
mac_econ_fwd500	na	7.1353
Cube_Coup_dt0	0.8591	1.7402
mcfe	0.8580	2.5414
webbase-1M	na	1513.4339
roadNet-PA	0.9264	4.2451
dc1	na	17407.9570
thermal2	1.0947	2.5041
cavity10	0.8546	2.1084
cant	1.2048	1.2276
FEM_3D_thermal1	0.9989	1.1208
raefsky2	0.5937	1.1898
mhd4800a	1.0352	1.5491
cop20k_A	0.3263	2.1354

Table 2: Speed-up del formato ELLPACK confrontata con il rapporto tra i dati salvati e i non zeri