Report
Sistemi di Calcolo Parallelo e Applicazione

Valerio Cristofori

AA 2021/2022

Abstract

In questo documento vengono argomentate le scelte effettuate per realizzare un nucleo di calcolo per il prodotto tra una matrice sparsa e un vettore.
Vengono poi presentati e analizzati i risultati del calcolo.

1 Introduzione

L'obiettivo è realizzare un nucleo di calcolo parallelo differenziando 2 tipi di parallelizzazione. La prima è basata su un'implementazione del concetto di multithreading su sistemi a memoria condivisa: *OpenMP*. La seconda è *CUDA*, ovvero un'API che permette l'uso di *GPU* per il calcolo parallelo (general-purpose computing su *GPU*). Ogni tipo di parallelizzazione viene poi confrontata con l'implementazione di un nucleo di calcolo seriale.
La fase di calcolo è preceduta da una fase di preprocessamento e memorizzazione dal formato `MatrixMarket` ai formati di memorizzazione per matrici sparse: `CSR` e `ELLPACK`.
Le matrici usate per il test sono alcune matrici della "Suite Sparse Matrix Collection", disponibili all'indirizzo `https://sparse.tamu.edu/`.
È stata eseguita infine una fase di analisi delle prestazioni utile per:

1. Confrontare i vari formati di memorizzazione e capire quale dei due è più utile in determinate situazioni.

2. Confrontare i differenti nuclei di calcolo su *CPU* e *GPU*.

3. Valutare il livello di parallelizzazione che porta alle migliori performance.

2 Preprocessamento

2.1 Caricamento e Formattazione

Sono state usate le funzioni di libreria fornite all'indirizzo `http://math.nist.gov/MatrixMarket/` per caricare le matrici dal formato `MatrixMarket`, dove vengono specificati il numero di righe *M*, numero di colonne *N*, e il numero di nonzeri *NZ*.
Da notare che per le matrici simmetriche viene memorizzato su file solo un triangolo (superiore/inferiore), di conseguenza, in memoria, vengono ricostruiti gli array di indici di riga, colonna e valori dell'intera matrice, e aggiornato il numero dei *NZ*.
È stato aggiunto un controllo sulle matrici memorizzate come *"pattern"*, popolando il vettore relativo ai valori nonzero di coefficienti di valore 1.0.

Sono stati formati in memoria i seguenti vettori:

I(1:NZ): Vettori degli indici di riga dei coefficienti nonzero

J(1:NZ): Vettori degli indici di colonna dei coefficienti nonzero

val(1:NZ): Vettori dei valori dei coefficienti nonzero

Viene poi costruito un vettore di indici **idxs** della matrice, in modo tale che l'i-esima posizione sia relativa a *I[i] * N + J[i]*. Attraverso questo vettore vengono ordinati i vettori per indici di riga (per ogni riga si ordina poi sugli indici di colonna di quella riga). In questo modo il vettore **idxs** risultante rappresenta gli indici dei valori nonzero della matrice, riga per riga.

2.2 Formattazione CSR

Compressed Storage by Rows. In questo caso la matrice viene formattata nel seguente modo:

IRP(1:M+1): Vettore dei puntatori all'inizio di ciascuna riga

JA(1:NZ): Vettore degli indici di colonna

AS(1:NZ): Vettore dei coefficienti

Viene costruito il vettore *IRP* partendo dal vettore *idxs*. È stato tenuto conto della possibile presenza di matrici con righe senza valori nonzero; queste matrici portavano ad un calcolo errato (con valori di *IRP* a -1, come default): in questo caso le righe sono state scartate, ed è stato aggiornato il valore del numero di righe *M*.

2.3 Formattazione ELLPACK

In questo caso la matrice viene formattata nel seguente modo:

JA(1:M,1:maxnz): array 2D di indici di colonna

AS(1:M,1:maxnz): array 2D di coefficienti

La formattazione *ELLPACK* differisce dalla tipologia di parallelizzazione:

2.3.1 Formattazione ELLPACK per OpenMP

Nel caso di formattazione *ELLPACK* per parallelizzazione con *OpenMP* il valore di *maxnz* è fisso e pari al massimo valore di nonzeri per riga, su tutte le righe. Una volta trovato il valore *maxnz* si popolano i vettori *JA* e *AS*.
Da notare che per ogni riga che conta un numero di nonzeri minore di *maxnz* i rispettivi coefficienti sono nulli per *AS* e sono uguali all'ultimo indice valido incontrato lungo la riga per *JA*.

2.3.2 Formattazione ELLPACK per CUDA

Per quanto riguarda *CUDA* invece, oltre a trovare il valore *maxnz* e inizializzare i vettori *JA* e *AS*, viene costruito il vettore **MAXNZ(1:M)**, vettore di numeri di nonzeri per ogni riga.
Successivamente viene eseguita la trasposizione degli array 2D *JA* e *AS*. Rendendo trasposti i dati i coefficienti di ciascuna colonna risultano adiacenti in memoria, essendo *C* un linguaggio con memorizzazione per righe.

3 Nucleo di calcolo

Sono state implementate differenti funzioni responsabili di eseguire il calcolo del prodotto in modo parallelo (con `OpenMP` e `CUDA`) e in modo seriale, senza parallelizzazione.
Nel primo caso sono stati distinti 2 nuclei per tipologia di parallelizzazione: uno per il calcolo del prodotto con matrice formattata secondo il formato *CSR* e il secondo con matrice formattata secondo *ELLPACK*.
Nel caso di calcolo seriale è stato scelto di implementare un unico nucleo con matrice *CSR*.
In tutti i casi il prodotto è stato effettuato con un vettore di coefficienti random compresi tra 0.1 e 3, salvato su file.
Infine, per il controllo sul risultato del prodotto, il vettore risultante è stato confrontato con il vettore *res_seq*, costruito come segue:

```
for (i = 0; i < mat->nz; i++)
  {
    int rInd = mat->I[i];
    int cInd = mat->J[i];
    res_seq[rInd] += mat->val[i] * vec->X[cInd];
  }
```

Il vettore *res_seq* è stato calcolato con l'unico fine quello di verificare le funzionalita' dei nuclei implementati, e non le performance.

3.1 Calcolo Seriale

Per il calcolo del prodotto in modo seriale si è scelta come unica formattazione quella *CSR*, quindi risulta:

```
for (int i = 0; i < M; i++)
    {
        for (int j = IRP[i]; j <= IRP[i+1] - 1; j++)
        {
          int tmp = JA[j];
            res[i] += AS[j] * X[tmp];
        }
    }
```

3.2 Calcolo con OpenMP

Vengono distinti 2 nuclei di calcolo, uno per tipo di formattazione scelta.
In ogni caso si è scelto uno schedule di tipo *dynamic* in modo tale che ogni thread *OpenMP* esegue un blocco di iterazioni di dimensione *chunk-size*, e quindi richiede un altro blocco finché non ci sono più blocchi disponibili.

Non si è scelto uno schedule *static* perché non si richiede nessun tipo di ordinamento nella distribuzione dei *chunks* (sembra più utile in questo caso un approccio di tipo *FCFS*). Si è osservato che la *chunk-size* sembra non influire molto nei tempi di calcolo (se non per valori molto piccoli): si è scelto un valore di ordine 3.

3.2.1 Calcolo OpenMP con CSR

Codice *C* dell'implementazione *OpenMP* con *CSR*:

```
#pragma omp parallel num_threads(thread_num)
{
#pragma omp for private( j, ja ) schedule( dynamic, CHUNK_SIZE )
    for( int i = 0; i < M; i++ ){
        double result = 0.0;
        for( j = IRP[i]; j <= IRP[i+1] - 1; j++ ){

            ja = JA[j];
                result += AS[j] * X[ja];
        }
        res[i] = result;
    }
}
```

Si è scelto di distribuire i thread sulle righe della matrice in modo tale da eliminare il più possibile dipendenze tra threads.

3.2.2 Calcolo OpenMP con ELLPACK

Codice *C* dell'implementazione *OpenMP* con *ELLPACK*:

```
#pragma omp parallel num_threads(thread_num)
{
#pragma omp for private( j, ja ) schedule( dynamic, CHUNK_SIZE )
    for( int i=0; i < M; i++ ){
        double result = 0.0;
        for( j = 0; j < maxnz; j++ ){

            ja = JA[i*maxnz + j];
                result += AS[i*maxnz + j] * X[ja];
        }
        res[i] = result;
    }

}
```

Anche in questo caso sono stati assegnati i threads alle righe della matrice.

3.3 Calcolo con CUDA

Vengono distinti 2 nuclei di calcolo, uno per tipo di formattazione scelta.
In entrambi i casi si definisce una griglia 1D di blocchi di threads 1D come segue:

```
dim3 GRID_DIM((M - 1 + BLOCK_DIM.x)/ BLOCK_DIM.x ,1)
dim3 BLOCK_DIM(BLOCK_SIZE)
```

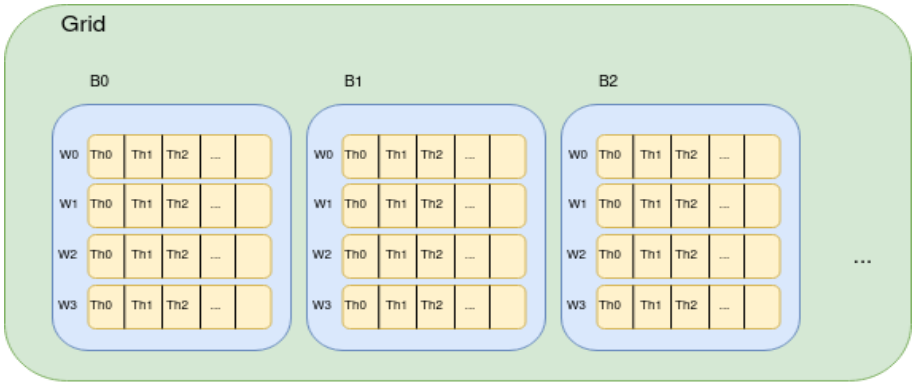

Figure 1: Grid 1D con block 1D: esempio con blocchi di dimesione 128 (4 warp)

3.3.1 Calcolo CUDA con CSR

Nel caso di formattazione *CSR* è stato implementato un kernel che assegna ad ogni riga della matrice un unico warp per il prodotto. I primi 2 thread del warp che sta processando una riga caricano, dal vettore *IRP*, gli indici di inizio e fine della riga corrente, e successivamente ogni thread del warp esegue il prodotto a salti di *WARPSIZE*.
Infine si esegue una `__syncthreads()` per sincronizzare tutti i thread di un warp per il successivo passo di riduzione (somma) di tutte le somme dei prodotti calcolati, per ciascun warp. Il thread 0 di ogni warp è poi responsabile di scrivere il risultato della riduzione nel vettore *results*.

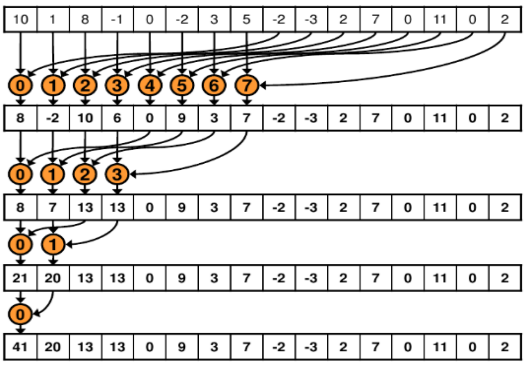

Figure 2: Ultimi 4 passi di riduzione per somma

Di seguito il kernel:

```
//iterate over rows
//every row is processed by a single warp
   for(int row = warp_id; row < num_rows; row += num_warps){
        // the first two threads of the warp are responsible
        // to fetch IRP[row] and IRP[row+1]
        if(thread_in_warp < 2)
            ptrs[warp_in_block][thread_in_warp] = IRP[row + thread_in_warp
    ]; //update the ptrs vector of row-start and row-end
        int row_start = ptrs[warp_in_block][0];
        int row_end   = ptrs[warp_in_block][1];

        // compute local sum
        double sum = 0;
        for(int j = row_start + thread_in_warp; j < row_end; j += WARP_SIZE
    )
        {
            sum += AS[j] * X[JA[j]];
        }

        volatile double* smem = sdata;
        smem[threadIdx.x] = sum; __syncthreads(); // reduction
        smem[threadIdx.x] = sum = sum + smem[threadIdx.x + 16];
        smem[threadIdx.x] = sum = sum + smem[threadIdx.x +  8];
        smem[threadIdx.x] = sum = sum + smem[threadIdx.x +  4];
        smem[threadIdx.x] = sum = sum + smem[threadIdx.x +  2];
        smem[threadIdx.x] = sum = sum + smem[threadIdx.x +  1];

        // first thread in warp
        // writes warp result
        if (thread_in_warp == 0){
            results[row] = smem[threadIdx.x];
        }
 }
```

Il passo di riduzione fa in modo di calcolare l'i-esima componente del vettore *results*. Ciò è reso possibile dal tipo *volatile* del vettore *smem*: in questo caso una scrittura su *smem* aggiorna effettivamente la memoria e non lascia che il compilatore ottimizzi caricando un valore salvato in precedenza in un registro.

3.3.2 Calcolo CUDA con ELLPACK

Nel caso di formattazione *ELLPACK* si è deciso di assegnare il prodotto di una riga della matrice a un singolo thread.
Il singolo thread esegue quindi un prodotto tra il vettore e la riga della matrice che è stata però trasposta in precedenza. Di seguito il kernel:

```
    const int num_rows = numRows;
    int           maxnz = MAXNZ[row];
    double          dot = 0;
    int             col = -1;
    double          val = 0;
    int               i = 0;
    for(i = 0; i < maxnz; i++)
    {
        col = JA[num_rows*i+row];
        val= AS[num_rows*i+row];
        dot += val*X[col];
    }
    return dot;
```

La trasposizione è stata effettuata per ottenere una buona *memory coalescing* perché, avendo assegnato un thread per riga, gli accessi coalizzati richiedono un layout trasposto in memoria: quando ogni thread legge una colonna di dati in memoria, ad ogni istruzione di read i threads adiacenti nel warp caricano i dati adiacenti dalla memoria; mentre nel caso in cui un thread leggesse una riga dalla memoria i threads adiacenti nel warp caricherebbero dati non adiacenti, ma lontani di un offset *row-size* (meno efficiente).

4 Test e Performance

Dal file *.csv* risultato dell'esecuzione dei test sono stati analizzati i dati relativi alle performance di calcolo per tutti i nuclei implementati. Sono stati calcolati i tempi di esecuzione per 3 iterazioni successivi di ogni nucleo, così da ricavare un tempo medio di esecuzione per nucleo.
Per la fase di testing sono stati implementati 2 script `bash`, *omp-tests.sh* e *cuda-tests.sh*: il primo esegue, per ogni matrice di test, il modulo di calcolo seriale e il calcolo parallelo con *OpenMP* con un numero di thread variabile da 1 al numero di core della macchina; il secondo esegue, per ogni matrice di test, il modulo di calcolo con *CUDA* con la dimensione del blocco di threads variabile da 128 a 1024, a salti di 128.
I test sono stati costruiti anche per valutare quale sia il miglior valore di threads per il calcolo.
Sono state calcolate le seguenti quantità per analizzare le performance:

$$Speedup_{cpu/gpu} = \frac{T_{serial}}{T_{cpu/gpu}} \tag{1}$$

$$Speedup_{cpu-gpu} = \frac{T_{cpu}}{T_{gpu}} \tag{2}$$

$$GFlops_{cpu/gpu} = \frac{2 * NZ}{T_{cpu/gpu}} \tag{3}$$

4.1 Analisi numero thread per CPU

Dai test effettuati per *OpenMP* risulta che le performance migliori si raggiungono, per la maggior parte delle matrici, con l'uso di 20 threads.
Questo si può spiegare dal fatto che meno di 20 threads non rendono efficace la parallelizzazione (non si usano pienamente le risorse date dalla *CPU*), mentre l'uso di più di 20 threads rallenta le prestazione. Il rallentamento è dato dal collo di bottiglia causato dalla memoria: risulta che il processore considerato per il test (*Intel Xeon Silver 4210 CPU @ 2.20GHz*) ha 2 sockets con 10 cores reali ciascuno, di conseguenza il limite a 20 threads è dettato dalla quantita' di cores effettivi della macchina (oltre i quali i thread in aggiunta possono saturare il bus della memoria e decrementare le prestazioni).
Tutto ciò si può osservare nei grafici riportati nelle figure 3 e 4.

4.2 Analisi dimensione blocco thread per GPU

Dai test effettuati per *CUDA*, come si può osservare in figura 5 e 6, al variare della dimensione dei blocchi i tempi di calcolo e i Gigaflops non cambiano.
In molti casi, soprattutto nel formato *CSR*, risulta che la dimensione del blocco che porta a performance migliori è la minima dimensione testata: 128 threads. Ma, anche in questo caso, i Gigaflops non cambiano di molto.

4.3 Analisi performance per CPU

Sono stati effettuati test per misurare lo *Speedup* del tempo di calcolo parallelo rispetto a quello seriale, per tutte le matrici.
Nelle figure 7 e 8 sono riportati gli istogrammi dello *Speedup* del calcolo in *CPU* sia per il caso *CSR* che per *ELLPACK*.
Per ogni matrice sono stati eseguiti 4 test relativi all'uso di 4, 8, 16, e 32 threads.
Come già osservato, nella maggior parte dei casi (per le matrici più grandi), l'uso di 16 threads porta a prestazioni migliori, mentre per la matrici molto piccole si favorisce l'uso di 4 threads e per quelle di medie dimensioni 8 threads.
Confrontando le performance per i due formati di memorizzazione si osserva che per il formato *CSR* i valori di *Speedup* risultano visibilmente più grandi rispetto al formato *ELLPACK*: questo può essere dovuto al fatto che le matrici sbilanciate (ovvero matrici per le quali alcune righe hanno un numero di nonzeri fortemente superiore del valore di nonzeri delle altre righe) portano ad una memorizzazione di una quantita' di zeri elevata, ciò implica anche che le matrici `dc1` e `webbase-1M` non sono state processabili con questo formato.
Non è stato possibile, infatti, adottare la strategia di calcolo attraverso il vettore di numeri di nonzeri per riga *MAXNZ(1:M)* (adottata nel caso di *CUDA*).

4.4 Analisi performance per GPU

Nelle figure 9 e 10 si riportano gli istogrammi delle performance su *GPU* dove, anche in questo caso, per ogni matrice sono stati effettuati i test per 128, 256, 512 e 1024 threads per blocco.
In questo caso le 4 misurazioni per ogni matrice sembrano essere simili nel caso *ELLPACK*, mentre sembra essere piu' conveniente il blocco di 128 per il caso *CSR*.
Confrontando i valori di *Speedup* nei due formati di memorizzazione risulta essere più vantaggioso adottare il formato *ELLPACK*, tranne per il fatto che, anche in questo caso, non è stato possibile effettuare il calcolo per le matrici `dc1` e `webbase-1M`. Le performance

decisamente migliori per il caso *ELLPACK* si hanno quando si stanno trattando matrici bilanciate, in cui il numero di nonzeri è circa lo stesso per ogni riga.

4.5 Confronto CPU-GPU e Conclusioni

In ultimo si vanno a confrontare le prestazioni calcolando lo *Speedup* della *GPU* rispetto alla *CPU*, ottenendo i risultati nelle figure 11 e 12.
Si nota che per entrambi i formati, per ogni matrice, le performance di calcolo sono migliori nel caso della *GPU*: tutti i valori infatti superano di gran lunga il valore di soglia settato a 1.
La grande differenza di tempi si conta soprattutto nel formato *ELLPACK*.

Dopo tutte le analisi effettuate, si conclude che per la maggior parte delle matrici (matrici di grandi dimensioni) è più vantaggioso utilizzare il nucleo di calcolo con *CUDA* con formattazione *ELLPACK*.
In generale, essendo il prodotto tra matrice sparsa e vettore un problema *memory bound*, le misure risultano soddisfacenti. Il collaudo è stato effettuato su una *Nvidia Quadro RTX 5000* dove la larghezza di banda risulta:

$$Bandwidth = Mem_{clockrate} * \frac{Mem_{buswidth}}{8} * 2 = 7001MHz * \frac{256bit}{8} * 2 = 448GB/s \quad (4)$$

Di conseguenza si può calcolare quanti sono i *Gflops/s* di picco, considerando di avere i dati da processare nella memoria della *GPU*:

$$Bandwidth * ArithmeticIntensity = 448GB/s * 0.25flops/Bytes = 112Gflops/s \quad (5)$$

dove *ArithmeticIntensity* definisce la quantita' di operazioni per byte letti da memoria (8 byte per la doppia precisione): *2*1flop/8bytes*.
Le performance misurate sono riportate nella tabella 5 e risultano adeguate per il problema considerato, i valori si distribuiscono tra un quarto e la metà dei Gigaflops di picco.

5 Tabelle e Grafici

	Speedup CPU	
Matrice	**CSR**	**ELLPACK**
adder_dcop_32	0.31	0.01
af_1_k101	10.17	9.15
af23560	5.51	5.15
amazon0302	8.71	8.70
bcsstk17	3.72	1.83
cage4	0.00	0.00
cant	8.62	7.81
cavity10	0.92	0.77
cop20k_A	8.97	2.73
Cube_Coup_dt0	10.27	8.13
dc1	4.18	-
FEM_3D_thermal1	4.57	4.09
lung2	5.95	4.49
mac_econ_fwd500	6.59	1.34
mcfe	0.50	0.31
mhd4800a	1.64	1.11
mhda416	0.22	0.19
ML_Laplace	10.31	9.21
nlpkkt80	10.08	9.04
olafu	7.37	4.09
olm1000	0.16	0.14
PR02R	9.39	4.92
raefsky2	2.24	1.75
rdist2	0.68	0.58
roadNet-PA	7.51	3.58
thermal1	4.73	5.46
thermal2	6.55	6.31
thermomech_TK	7.46	5.80
webbase-1M	6.80	-
west2021	0.23	0.22

Table 1: Speedup in CPU

	Speedup GPU	
Matrice	**CSR**	**ELLPACK**
adder_dcop_32	1.40	0.91
af_1_k101	69.18	117.62
af23560	34.48	73.42
amazon0302	22.23	106.82
bcsstk17	29.95	43.39
cage4	0.03	0.03
cant	88.96	91.34
cavity10	8.33	19.48
cop20k_A	60.38	87.71
Cube_Coup_dt0	111.98	121.24
dc1	8.23	-
FEM_3D_thermal1	31.26	70.94
lung2	11.00	38.16
mac_econ_fwd500	17.96	36.84
mcfe	2.97	6.49
mhd4800a	10.80	26.38
mhda416	1.26	3.14
ML_Laplace	118.18	120.66
nlpkkt80	71.57	114.54
olafu	56.46	87.36
olm1000	0.63	1.93
PR02R	87.83	69.30
raefsky2	21.07	60.16
rdist2	5.85	14.00
roadNet-PA	14.82	111.95
thermal1	18.56	55.68
thermal2	29.37	110.83
thermomech_TK	22.36	67.73
webbase-1M	12.62	-
west2021	0.83	2.78

Table 2: Speedup in GPU

	GFlops CPU	
Matrice	**CSR**	**ELLPACK**
adder_dcop_32	0.04	0.00
af_1_k101	4.99	4.49
af23560	2.32	2.17
amazon0302	2.17	2.28
bcsstk17	1.77	0.66
cage4	0.00	0.00
cant	3.77	2.97
cavity10	0.38	0.25
cop20k_A	3.34	0.99
Cube_Coup_dt0	5.09	4.06
dc1	1.13	-
FEM_3D_thermal1	1.89	1.83
lung2	2.25	1.76
mac_econ_fwd500	2.44	0.49
mcfe	0.16	0.10
mhd4800a	0.60	0.43
mhda416	0.08	0.07
ML_Laplace	5.04	4.54
nlpkkt80	4.91	4.41
olafu	2.92	2.17
olm1000	0.04	0.04
PR02R	4.78	2.43
raefsky2	0.75	0.65
rdist2	0.30	0.21
roadNet-PA	1.79	0.86
thermal1	2.25	1.76
thermal2	1.99	1.85
thermomech_TK	2.21	1.86
webbase-1M	1.91	-
west2021	0.07	0.06

Table 3: GIGAFlops in CPU

	GFlops GPU	
Matrice	**CSR**	**ELLPACK**
adder_dcop_32	0.40	0.26
af_1_k101	33.41	56.80
af23560	16.05	34.18
amazon0302	5.45	26.18
bcsstk17	13.98	20.25
cage4	0.01	0.01
cant	41.89	43.01
cavity10	3.50	8.18
cop20k_A	22.30	32.40
Cube_Coup_dt0	54.10	58.58
dc1	3.41	-
FEM_3D_thermal1	14.60	33.13
lung2	4.35	15.08
mac_econ_fwd500	6.96	14.28
mcfe	1.13	2.48
mhd4800a	4.83	11.80
mhda416	0.49	1.22
ML_Laplace	55.83	56.99
nlpkkt80	34.17	54.68
olafu	26.60	41.16
olm1000	0.20	0.60
PR02R	41.55	32.78
raefsky2	9.98	28.48
rdist2	2.55	6.10
roadNet-PA	3.65	27.53
thermal1	6.57	19.70
thermal2	8.99	33.94
thermomech_TK	6.65	20.14
webbase-1M	3.96	-
west2021	0.29	0.98

Table 4: GIGAFlops in GPU

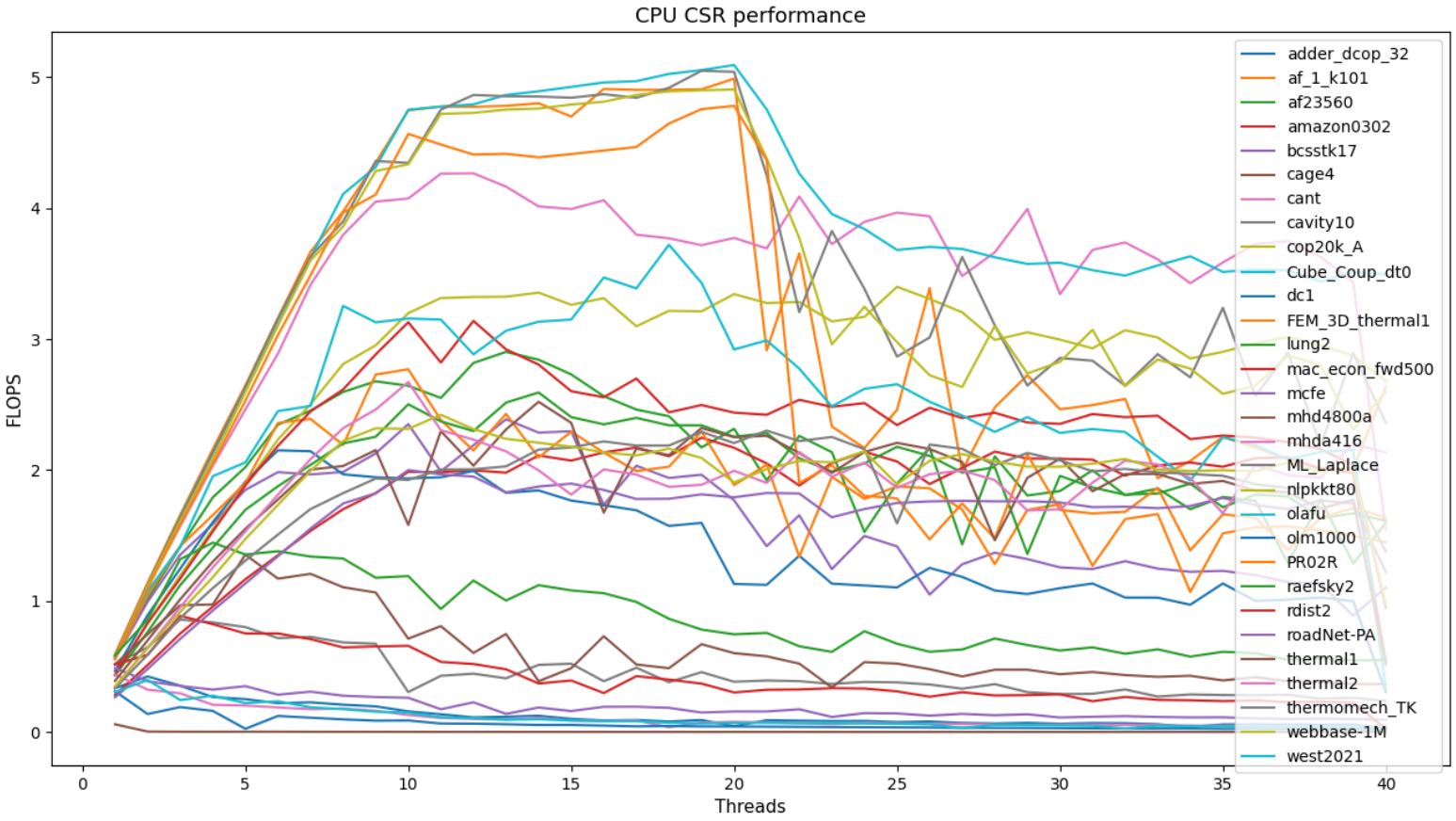

Figure 3: GFLOPS al variare dei threads nel calcolo su CPU con formato CSR

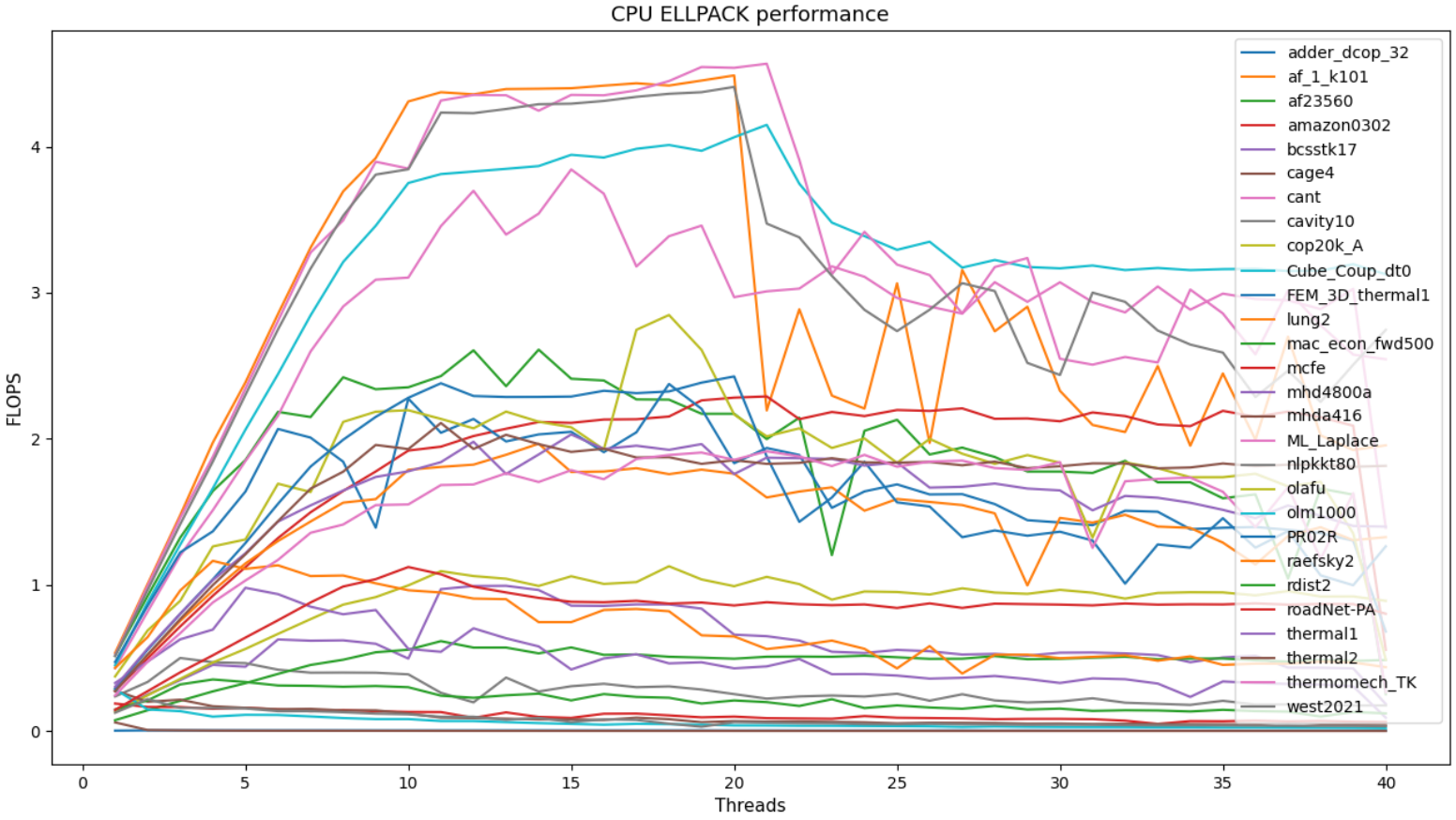

Figure 4: GFLOPS al variare dei threads nel calcolo su CPU con formato ELLPACK

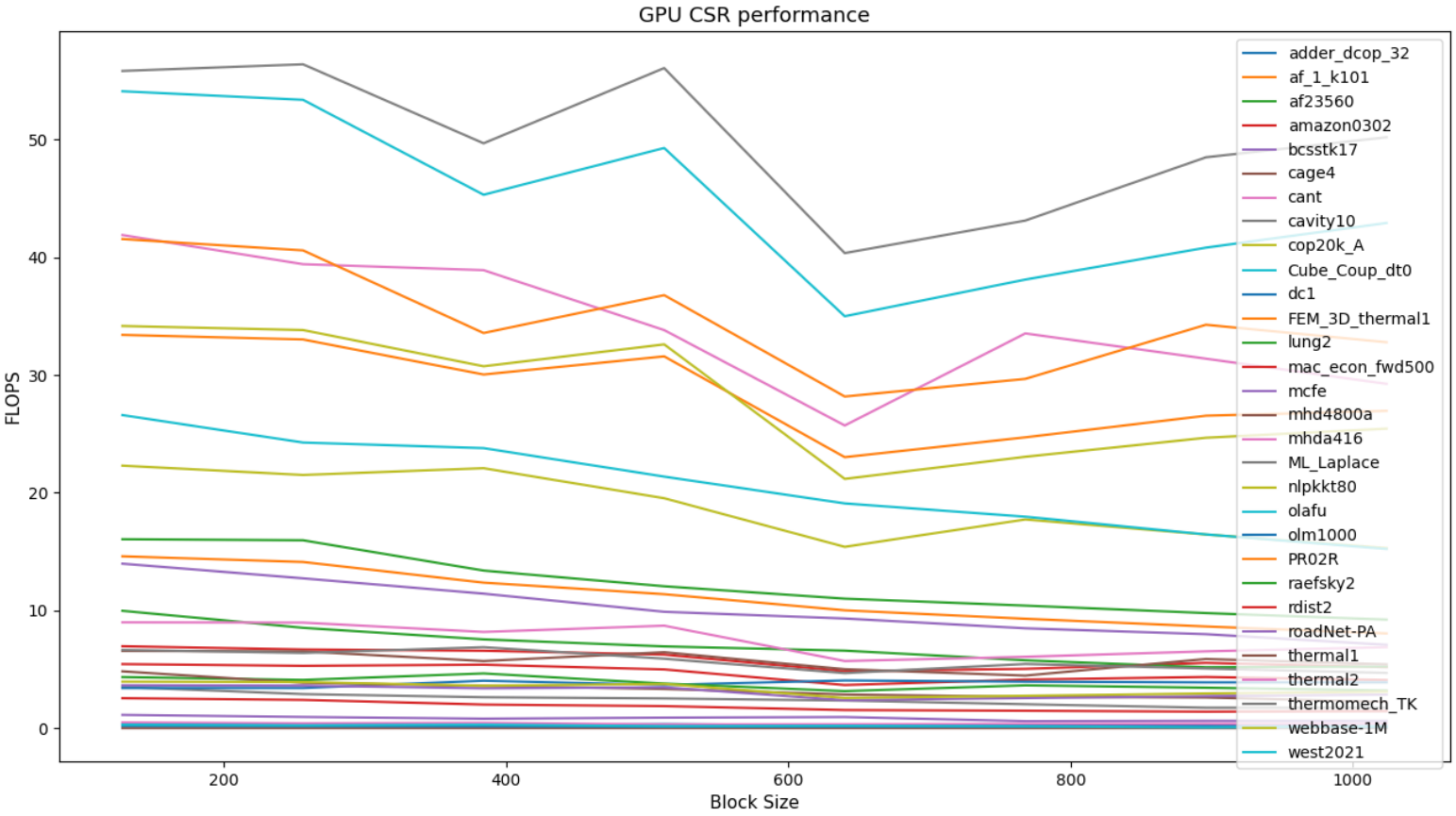

Figure 5: GFLOPS al variare della dimensione del blocco nel calcolo su GPU con formato CSR

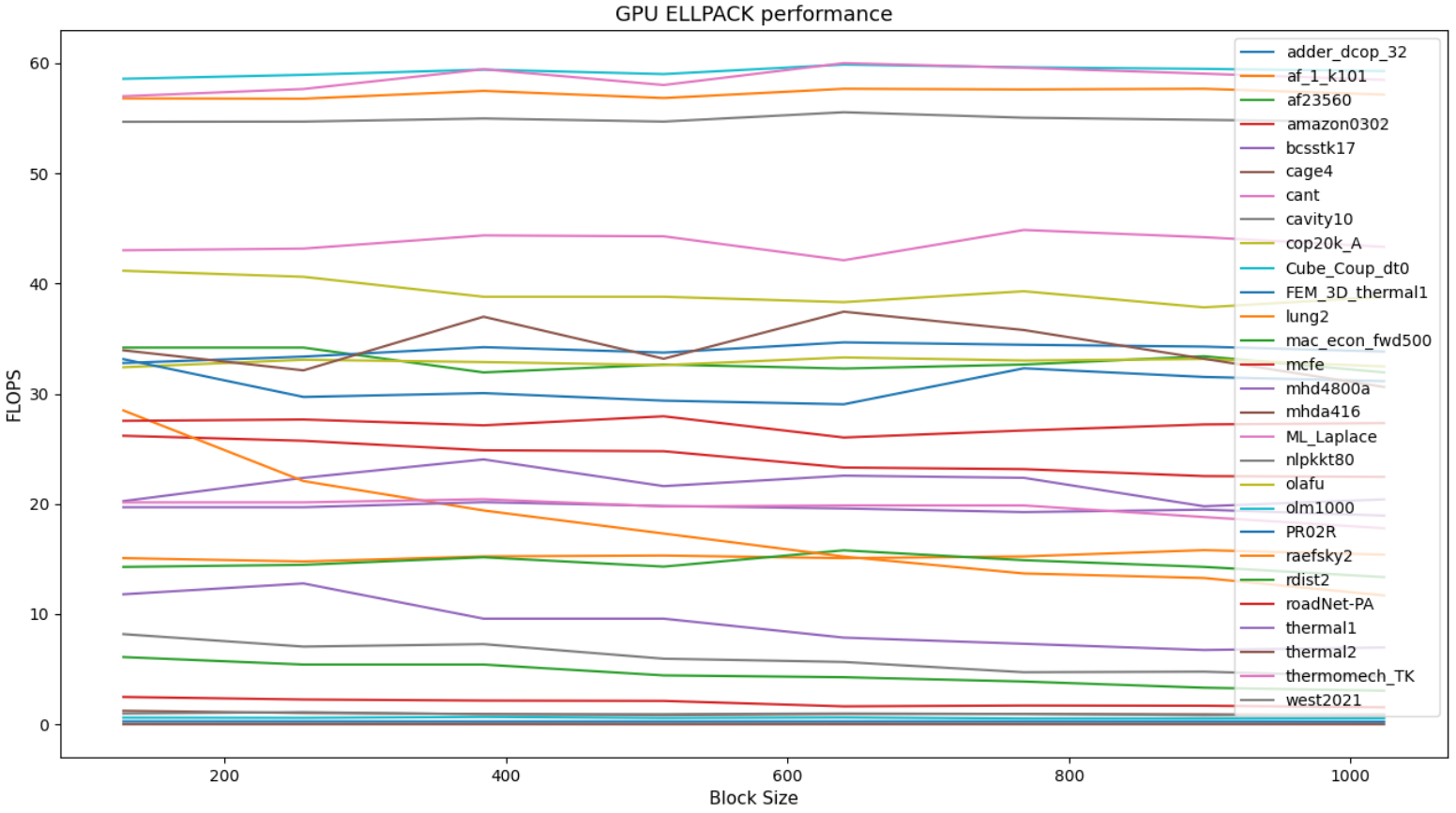

Figure 6: GFLOPS al variare della dimensione del blocco nel calcolo su GPU con formato ELLPACK

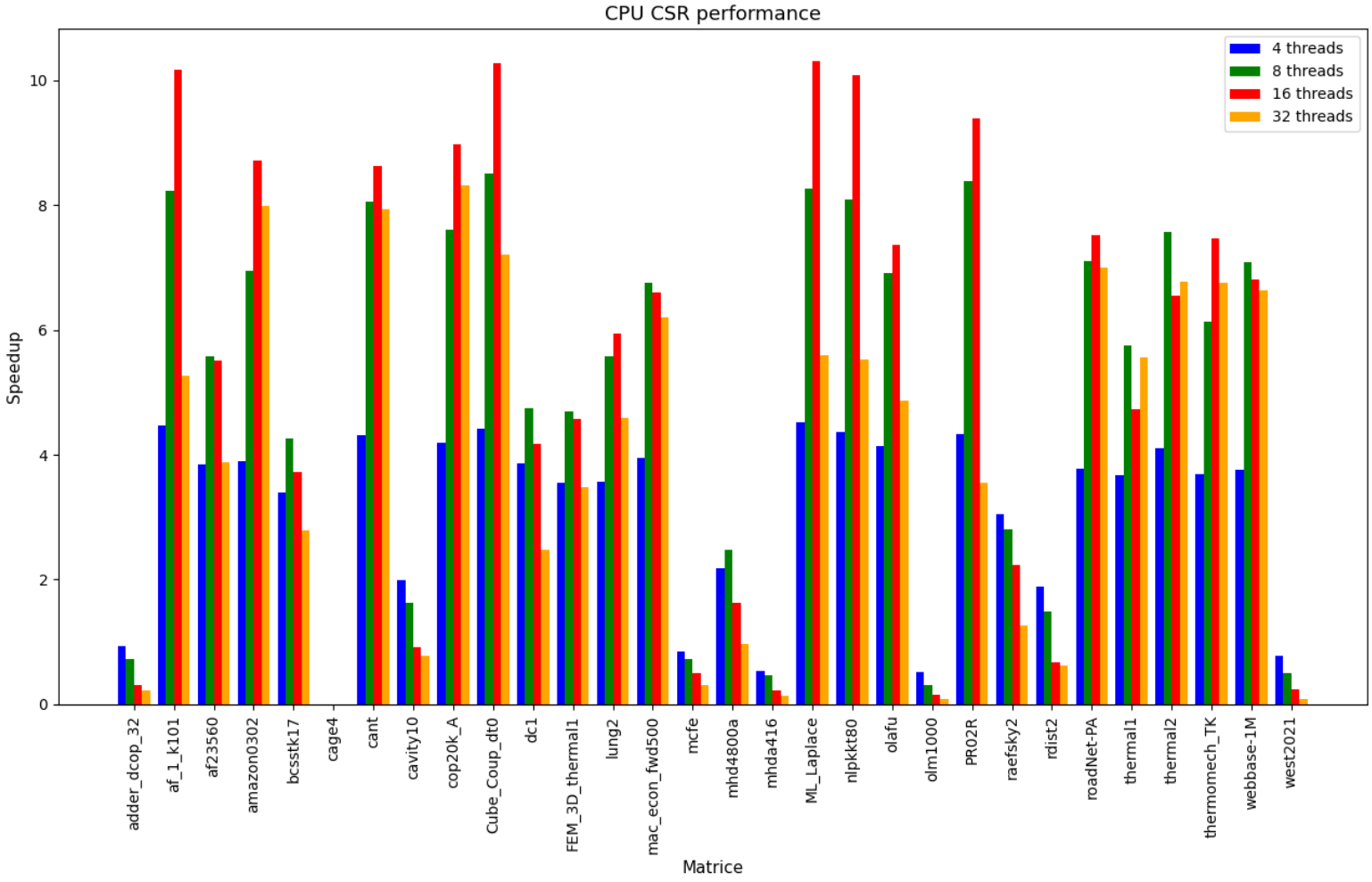

Figure 7: Speedup calcolo su CPU con formato CSR

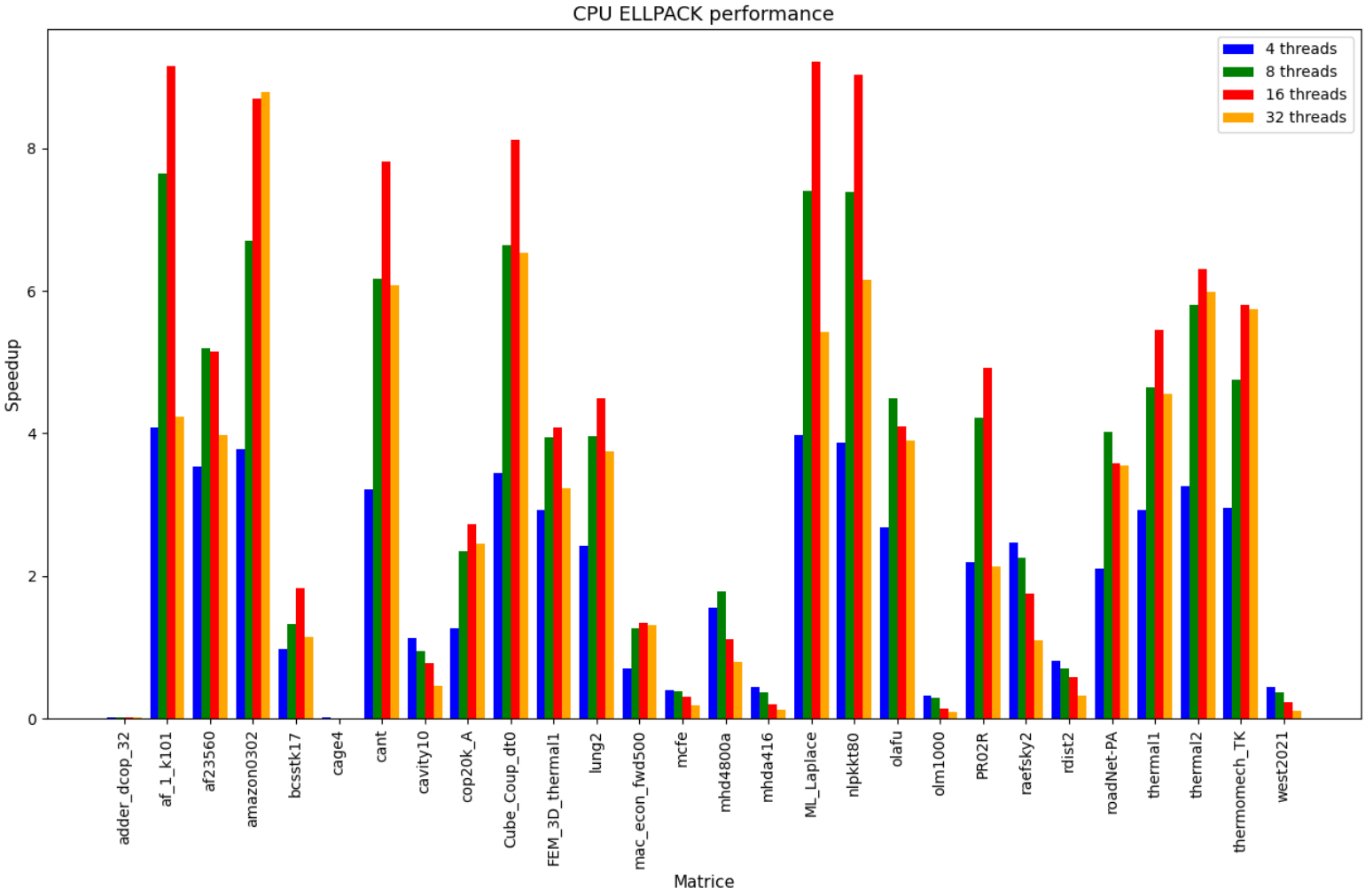

Figure 8: Speedup calcolo su CPU con formato ELLPACK

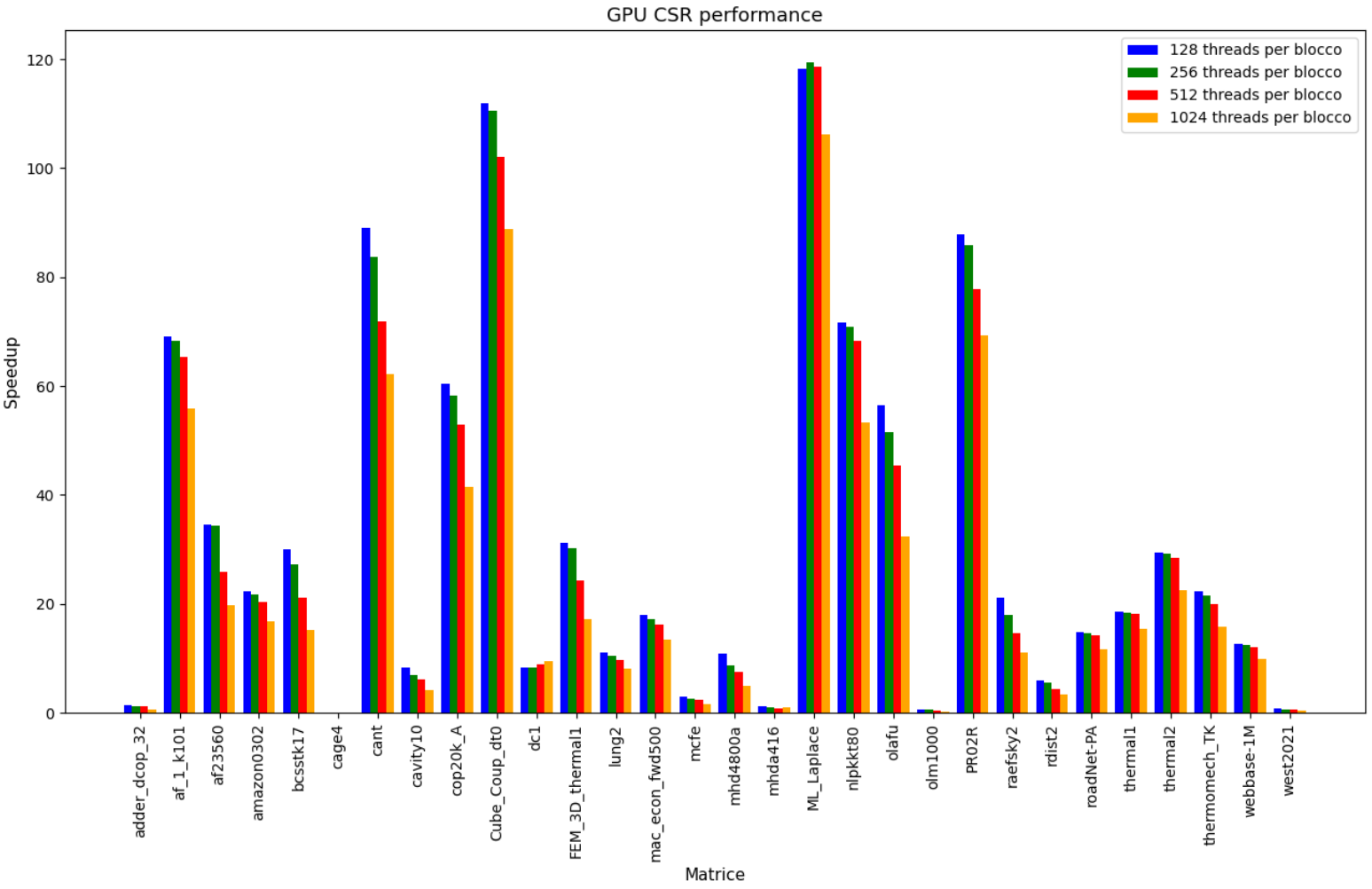

Figure 9: Speedup calcolo su GPU con formato CSR

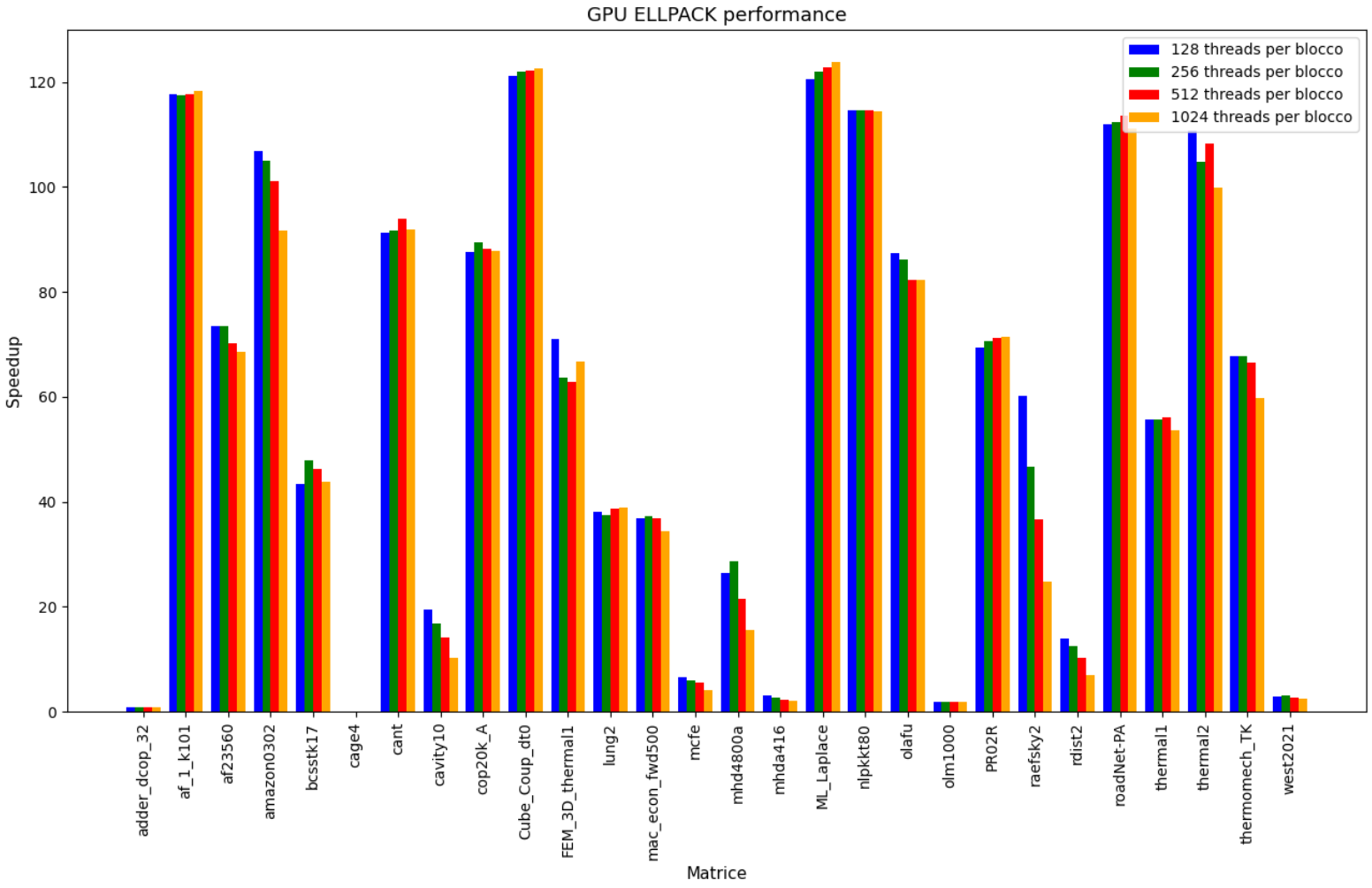

Figure 10: Speedup calcolo su GPU con formato ELLPACK

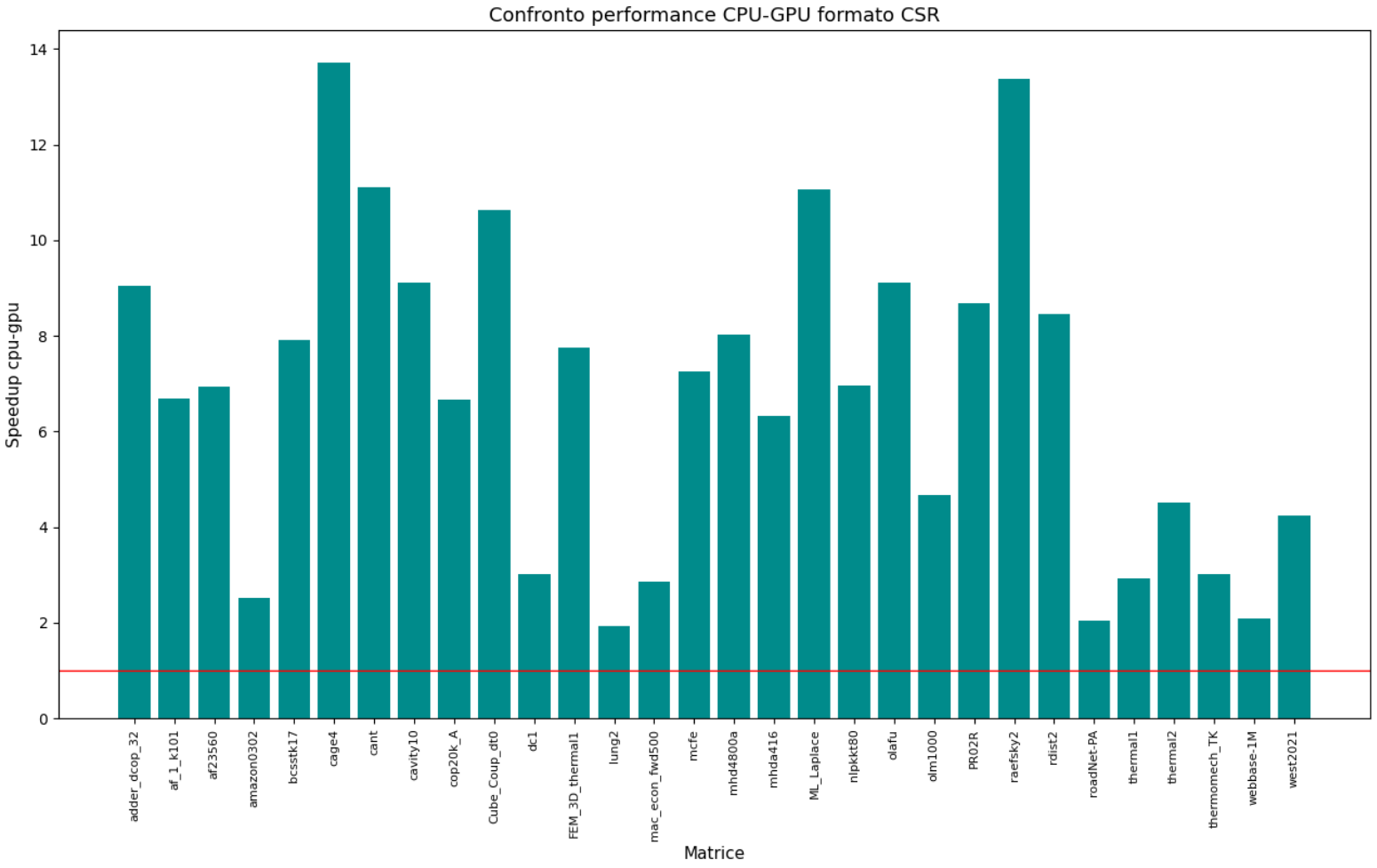

Figure 11: Speedup tra CPU e GPU del calcolo con formato CSR

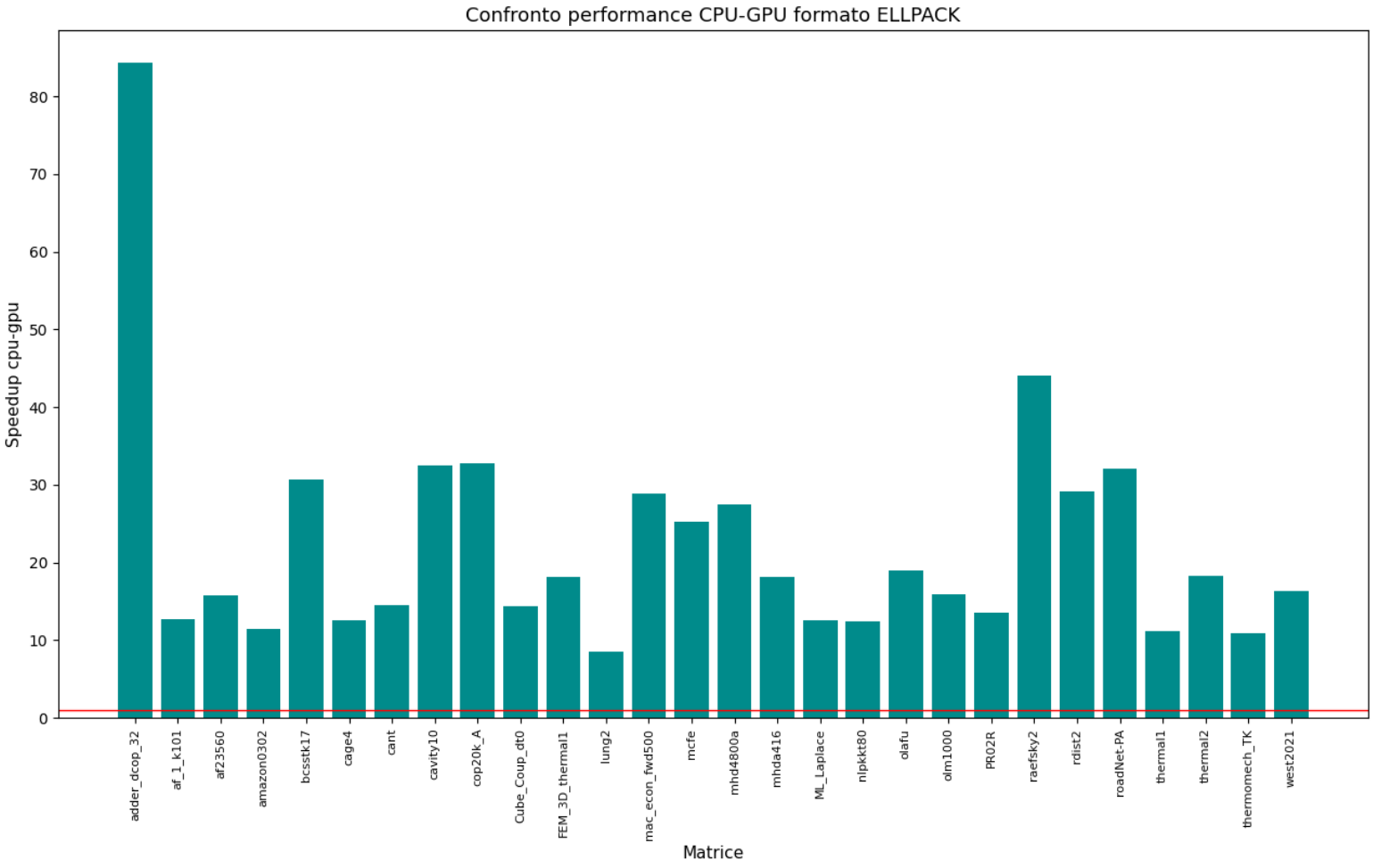

Figure 12: Speedup tra CPU e GPU del calcolo con formato ELLPACK