Prodotto SpMV con vettore denso

Matteo Calzetta, Alessandro Lori

April 11, 2025

Task

Il problema è crare un nucleo di calcolo parallelizzato che svolga il prodotto $y = Ax$, dove A è una matrice sparsa.

Per il formato di memorizzazione della matrice sono stati usati i formati:

- CSR (Compressed sparse row)
- HLL (Hacked ellpack)

Il prodotto è stato eseguito in maniera seriale e poi parallelizzato, i risultati confrontati con l'implementazione seriale.

Data collection

I dati sono stati raccolti sul server di dipartimento:

Componente	**Specifica**
Processore	Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz
GPU	NVIDIA RTX5000

Sistema: Dual socket con 20 thread fisici e 40 thread logici.

Preprocessing

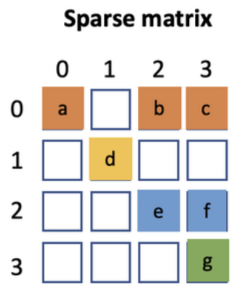
La conversione delle matrici dal formato Matrix Market (`.mtx`) avviene tramite la libreria ANSI C per Matrix Market I/O. Questo consente di identificare strutture speciali considerate (*Pattern*, *Symmetric*, *Array*) e ricostruire correttamente la matrice.

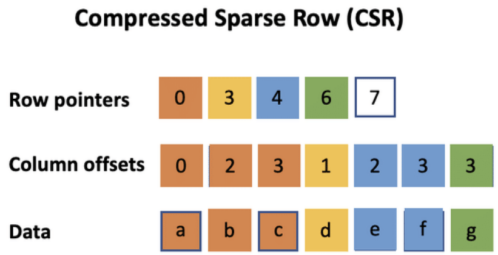
Una volta letti i valori non nulli, essi vengono salvati nei formati CSR e HLL per ottimizzare sia il consumo di memoria sia le successive operazioni di calcolo.

CSR (Compressed Sparse Row)

Della matrice vengono memorizzati solo gli elementi non nulli e i relativi indici:

- **AS**: array dei valori non nulli.
- **JA**: array degli indici colonna dei valori non nulli.
- **IRP**: array di puntatori al primo elemento di ciascuna riga in `AS`.





HLL (Hacked ELLPACK)

Hacked ELLPACK: dividere la matrice in blocchi (hack) e memorizzare ciascun blocco come una mini-matrice in formato ELLPACK, riducendo il padding. Ogni blocco memorizza:

- **MAXNZ**: numero massimo di non zeri per riga nel blocco.
- **JA**: array bidimensionale per gli indici di colonna.
- **AS**: array bidimensionale per i valori non nulli.

3	0	1	0
0	0	0	0
0	2	4	1
1	0	0	1

(a) Matrix

values:

3	1
0	0
2	4
1	1

column indices

0	2
*	*
1	2
0	3

(b) Arrays

HLL Column-Major

Per ottimizzare l'accesso alla memoria in GPU, abbiamo adottato una variante *column-major* di HLL, dove i valori non nulli di ogni blocco vengono memorizzati colonna per colonna. Per entrambi i formati inoltre, AS e JA sono array monodimensionali.

Vantaggi:

- Accessi più *coalescenti* in GPU, sfruttando al meglio la larghezza di banda della memoria.
- Semplifica la gestione memoria *host* e *device*.
- Evita frammentazione tipica degli array bidimensionali dinamici.

Esempio accesso column-major

Consideriamo un blocco con `rows = 3` e `maxNZ = 2`:

$$A = \begin{bmatrix} 5 & 0 \\ 0 & 2 \\ 3 & 4 \end{bmatrix}$$

Memorizzata in column-major:

- Colonna 0: $AS = [5,\ -,\ 3] \quad JA = [0,\ -,\ 0]$
- Colonna 1: $AS = [-,\ 2,\ 4] \quad JA = [-,\ 1,\ 1]$

Accesso all'elemento $r = 2,\ c = 1$:

$$\texttt{idx} = c \cdot \mathit{rows} + r = 1 \cdot 3 + 2 = 5$$

$$\Rightarrow \quad AS[5] = 4, \quad JA[5] = 1, \quad y[2] += 4 \cdot x[1]$$

CSR Seriale

Algorithm 1 Prodotto matrice sparsa-vettore con formato CSR

```
1: for i = 1 to m do
2:     t ← 0
3:     for j = irp(i) to irp(i + 1) − 1 do
4:         t ← t + as(j) · x(ja(j))
5:     y(i) ← t
```

Il ciclo esterno scorre le righe della matrice (M), il ciclo interno gli elementi non nulli e li moltiplica per il corrispondente del vettore denso x.

HLL row-major seriale

Algorithm 2 Prodotto matrice-vettore seriale con formato HLL row-major

```
1:  for blockID = 0 to num_blocks − 1 do
2:      start_row ← blockID · HackSize
3:      end_row ← min((blockID + 1) · HackSize, M)
4:      maxNZ ← hll_matrix.blocks[blockID].max_nz_per_row
5:      row_offset ← 0
6:      for i = start_row to end_row − 1 do
7:          y[i] ← 0
8:          for j = 0 to maxNZ − 1 do
9:              idx ← row_offset + j
10:             if JA[idx] ≠ −1 then
11:                 y[i] ← y[i] + AS[idx] · x[JA[idx]]
12:         row_offset ← row_offset + maxNZ
```

Il ciclo esterno scorre i blocchi della matrice, ogni blocco contiene un sottoinsieme di righe consecutive. Per ogni riga, si scorrono gli elementi non nulli (fino a maxNZ) e, se validi (non hanno padding -1), vengono moltiplicati per i corrispondenti valori del vettore denso x.

HLL column-major seriale

Algorithm 3 Prodotto matrice-vettore seriale con formato HLL column-major

```
1:  Inizializza y(i) ← 0 per i = 0 to M − 1
2:  for b = 0 to num_blocks − 1 do
3:      start_row ← b · HackSize
4:      rows ← HackSize
5:      if b = num_blocks − 1 and M mod HackSize ≠ 0 then
6:          rows ← M mod HackSize
7:      maxNZ ← hll_matrix.blocks[b].max_nz_per_row
8:      for r = 0 to rows − 1 do
9:          sum ← 0
10:         for c = 0 to maxNZ − 1 do
11:             idx ← c · rows + r
12:             val ← AS[idx]
13:             col ← JA[idx]
14:             sum ← sum + val · x[col]
15:         y[start_row + r] ← sum
```

Nel formato HLL **column-major**, gli elementi non nulli di ogni blocco sono memorizzati colonna per colonna. Per accedere all'elemento in posizione (`riga = r`, `colonna = c`) all'interno di un blocco, l'indice viene calcolato come:

$$\texttt{idx} = c \cdot \texttt{rows} + r$$

dove:

- `c` è la colonna logica all'interno del blocco (0-based)
- `r` è la riga logica all'interno del blocco (0-based)
- `rows` è il numero di righe effettive nel blocco corrente

Metriche di valutazione

Le prestazioni del prodotto matrice-vettore sono state misurate ripetendo più volte l'invocazione del kernel su ciascuna matrice, così da calcolare un **tempo medio** di esecuzione.

Sono state considerate le seguenti metriche:

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

$$\text{Speedup} = \frac{T_{\text{seriale}}}{T_{\text{parallelo}}}$$

$$\text{Efficienza} = \frac{\text{Speedup}}{\text{NumThreads}}$$

Validazione dei risultati

Per verificare la correttezza del prodotto matrice-vettore nei diversi formati e paradigmi (CSR, HLL row/column, OpenMP, CUDA), è stata calcolata la **norma L2** tra il risultato ottenuto e la versione seriale di riferimento:

$$\|\mathbf{a} - \mathbf{b}\|_2 = \sqrt{\sum_{i=0}^{M-1} (a_i - b_i)^2}$$

- Inizialmente confrontate le tre versioni sequenziali su matrici piccole.
- Successivamente, confrontate anche le versioni parallele (OpenMP/CUDA) con la rispettiva versione seriale.

La norma risultava prossima a zero in tutti i casi, confermando la correttezza. Piccole differenze numeriche sono attese a causa dell'aritmetica floating-point.

OpenMP con formato CSR

Per sfruttare il parallelismo su CPU con OpenMP, è stata implementata una strategia di bilanciamento del carico basata sulla distribuzione del numero di elementi non nulli tra i thread, mantenendo la località spaziale.

Strategia adottata:

- Le righe della matrice sono suddivise in blocchi contigui per thread.
- Ogni thread riceve un numero di non-zero bilanciato rispetto agli altri.
- Se un thread riceve un sottoinsieme vuoto, viene scartato.

Formula: si accumulano i non-zero riga per riga e si crea una nuova partizione quando si supera il carico aspettato

$$\texttt{target\_workload} = \frac{\texttt{total\_nnz}}{\texttt{num\_threads}}$$

OpenMP con formato CSR

```c
void csr_matvec_openmp(CSRMatrix *csr, double *x, double *y, int num_threads, int *row_partition) {
    #pragma omp parallel num_threads(num_threads)
    {
        int thread_id = omp_get_thread_num();
        int start_row = row_partition[thread_id];
        int end_row = row_partition[thread_id + 1];

        // Calcolo del prodotto matrice-vettore per il blocco assegnato
        for (int i = start_row; i < end_row; i++) {
            double sum = 0.0;
            for (int j = csr->IRP[i]; j < csr->IRP[i + 1]; j++) {
                y[i] += csr->AS[j] * x[csr->JA[j]];
            }
        }
    }
}
```

Ogni thread ottiene il proprio intervallo di righe tramite l'array row_partition, calcolato in fase di bilanciamento del carico. Per ciascuna riga assegnata, si esegue l'accumulo dei prodotti tra i valori non nulli (AS) e gli elementi corrispondenti del vettore x.

OpenMP con formato HLL

Per il prodotto matrice-vettore con formato HLL è stata usata la direttiva:

```
#pragma omp parallel for schedule(guided)
```

Caratteristiche principali:

- Lo scheduling `guided` assegna dinamicamente blocchi della matrice ai thread.
- Ogni thread elabora blocchi distinti, lavorando su righe disgiunte $\Rightarrow$ nessuna sincronizzazione necessaria.

Vantaggio: buon bilanciamento del carico anche in presenza di blocchi disomogenei.
Svantaggio: overhead introdotto dal monitoraggio e dalla riassegnazione dinamica dei blocchi.

CUDA CSR - Kernel 0: `spmv_csr_threads`

Assegnazione 1:1 tra thread e riga CSR. Ogni thread calcola il prodotto scalare di una riga.
Numero Blocchi: Ogni blocco ha 256 threads e il numero di blocchi lanciati viene calcolato attraverso la seguente formula che permette la copertura della matrice da parte dei blocchi con uno scarto in eccesso:

$$\texttt{num\_blocks} = \frac{M + \texttt{threads\_per\_block} - 1}{\texttt{threads\_per\_block}}$$

Calcolo indici:

$$\texttt{row} = \texttt{blockIdx.x} \cdot \texttt{blockDim.x} + \texttt{threadIdx.x}$$

Risultato: Ogni thread inserisce il risultato della computazione della riga all'interno del vettore risultante $\rightarrow$ nessuna race condition.

CUDA CSR - Kernel 1: `spmv_csr_warps`

Un warp per riga: i 32 thread di un warp collaborano per calcolare una riga.

Configurazione thread bidimensionale:

$$\texttt{blockDim.x} = 32, \quad \texttt{blockDim.y} = \texttt{warp\_per\_block}$$

$$\texttt{row} = \texttt{blockIdx.x} \cdot \texttt{blockDim.y} + \texttt{threadIdx.y}$$

Ogni thread lavora su:

```
for (j = row_start + lane; j < row_end; j += WARP_SIZE)
```

Riduzione con shuffle: `__shfl_sync` $\rightarrow$ somma dei risultati parziali all'interno del warp.

CUDA CSR - Kernel 2 e Kernel 3

Varianti: differiscono per il metodo usato per effettuare la riduzione rispetto al Kernel 1.

Kernel 2: usa la memoria condivisa nella quale sono salvati i risultati parziali

```
shared_sum[lane + threadIdx.y * WARP_SIZE] = sum
```

e successivamente ridotti all'interno di essa.

shared_sum[parziale_corrente] += shared_sum[parziale_corrente + offset]

Kernel 3: usa la memoria condivisa bidimensionale nella quale sono salvati i risultati parziali

```
shared_sum[warp_id][lane] = sum
```

```
sum = shared_sum[warp_id][lane]
```

e successivamente presi da essa e ridotti usando __shfl_sync.

```
sum += __shfl_sync(0xFFFFFFFF, sum, lane + offset)
```

CUDA CSR - Kernel 4: spmv_csr_warps_cachel2

Configurazione: abbiamo un warp per riga con stessa configurazione rispetto ai kernel precedenti.

CacheL2: usiamo __ldg per prendere i valori che ci interessano dalla CacheL2 prima di dover accedere alla memoria globale ove necessario.

```
int col = __ldg(JA[j]),  sum += __ldg(AS[j]) * __ldg(x[col])
```

Riduzione con shuffle: __shfl_sync $\rightarrow$ somma dei risultati parziali all'interno del warp.

CUDA CSR - Kernel 5: spmv_csr_warp_texture

Configurazione: abbiamo un warp per riga con stessa configurazione rispetto ai kernel precedenti.

Texture Cache: usiamo text1Dfetch per prendere i valori che ci interessano dalla Texture Cache prima di dover accedere alla memoria globale ove necessario.

```
float x_val = tex1Dfetch<float>(tex_x, JA[j])
```

Riduzione con shuffle: __shfl_sync $\rightarrow$ somma dei risultati parziali all'interno del warp.

CUDA – Kernel 0: `matvec_Hll_cuda_SH`

Assegnazione 1:1 tra thread e riga HLL. Ogni thread calcola il prodotto scalare di una riga.

Struttura HLL: blocchi da `HACK_SIZE` = 32 righe → allineamento con SM della GPU.

Calcolo indici:

$$\texttt{global\_row} = \texttt{blockIdx.x} \cdot \texttt{blockDim.x} + \texttt{threadIdx.x}$$

$$\texttt{block\_id} = \left\lfloor \frac{\texttt{global\_row}}{\texttt{HACK\_SIZE}} \right\rfloor, \quad \texttt{local\_row} = \texttt{global\_row} \bmod \texttt{HACK\_SIZE}$$

Scrittura sicura:

$$\texttt{d\_y[global\_row]} = \texttt{sum}$$

Ogni thread scrive in posizione esclusiva → nessuna race condition.

CUDA – Kernel 1: `matvec_Hll_cuda_warp`

Un warp per riga: i 32 thread di un warp collaborano per calcolare una riga.

Configurazione griglia bidimensionale:

$$\texttt{blockDim.x} = 32, \quad \texttt{blockDim.y} = \texttt{warp\_per\_block}$$

$$\texttt{global\_row} = \texttt{blockIdx.x} \cdot \texttt{blockDim.y} + \texttt{threadIdx.y}$$

Ogni thread lavora su:

```
for (j = lane; j < maxNZ; j += WARP_SIZE)
```

Riduzione con shuffle: `__shfl_down_sync` $\rightarrow$ somma dei risultati parziali all'interno del warp.

CUDA – Kernel 2: `matvec_Hll_cuda_warp_shared`

Riduzione con shared memory: i risultati parziali sono scritti in `sdata[]`.

Allocazione shared memory:

$$\texttt{sharedSize} = \texttt{WARP\_SIZE} \cdot \texttt{warps\_per\_block} \cdot \texttt{sizeof(double)}$$

Indice nella shared:

$$\texttt{s\_index} = \texttt{warpIdInBlock} \cdot \texttt{WARP\_SIZE} + \texttt{lane}$$

Riduzione logaritmica: somma tra coppie di valori con `offset` decrescente, facendo

$$\texttt{sdata[s\_index]} \mathrel{+}= \texttt{sdata[s\_index + offset]}$$

e

__syncthreads() ad ogni passo

CUDA – Kernel 3: `matvec_hll_column_kernel`

Formato column-major: massimizza coalescenza degli accessi in memoria globale.

Accesso column-major:

$$\texttt{idx} = c \cdot \texttt{rows} + \texttt{local\_row}$$

Esempio:

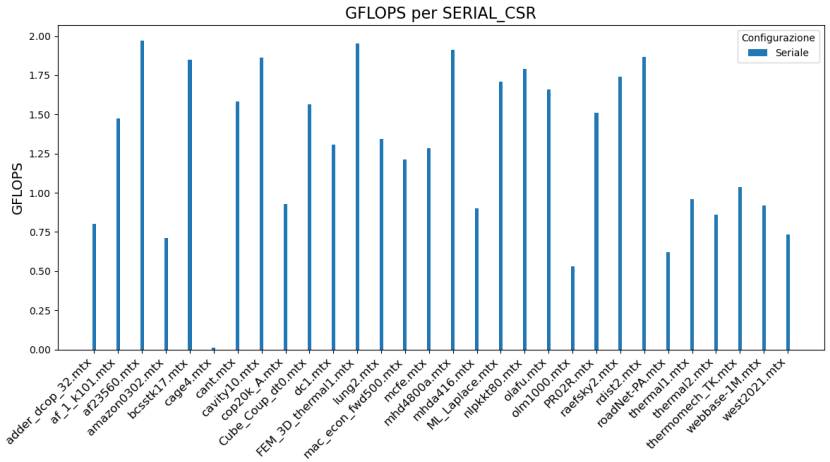
- Thread 0: R0_C0, R0_C1, R0_C2
- Thread 1: R1_C0, R1_C1, R1_C2
- ...

Coalescenza perfetta: accessi simultanei dei thread a celle contigue $\rightarrow$ efficienza massima.

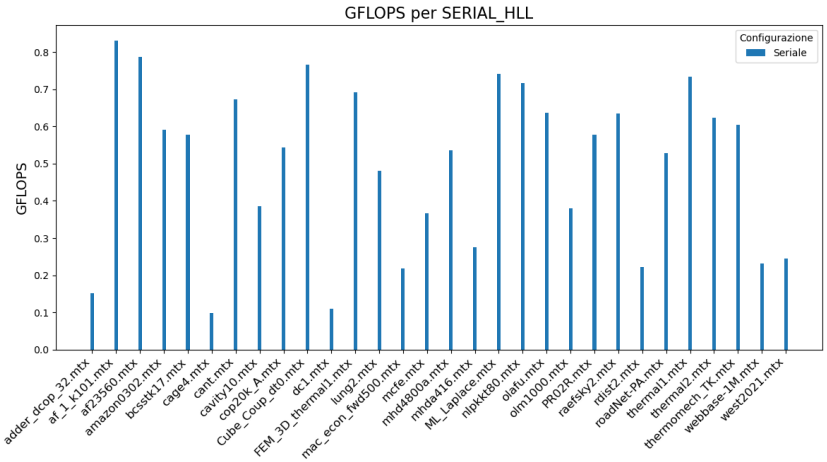
Scrittura esclusiva in:

$$\texttt{d\_y[global\_row]} = \texttt{sum}$$

Risultati seriale

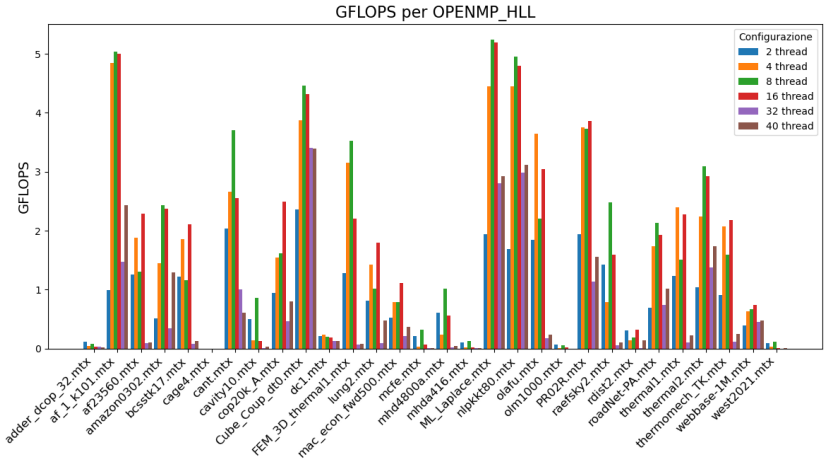


GFLOPS CSR seriale

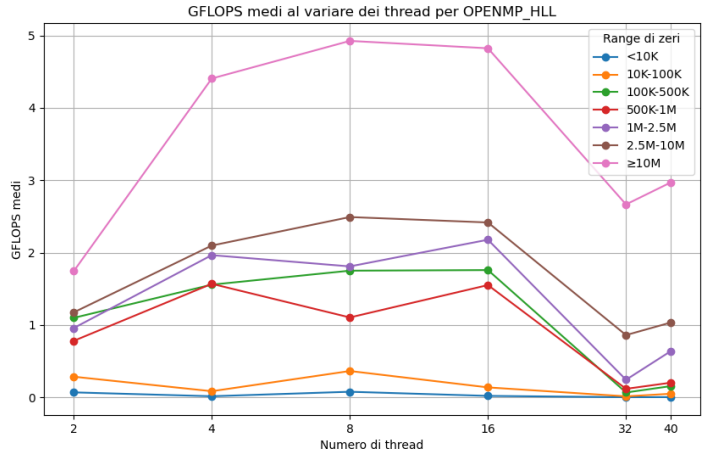


GLOPS HLL seriale

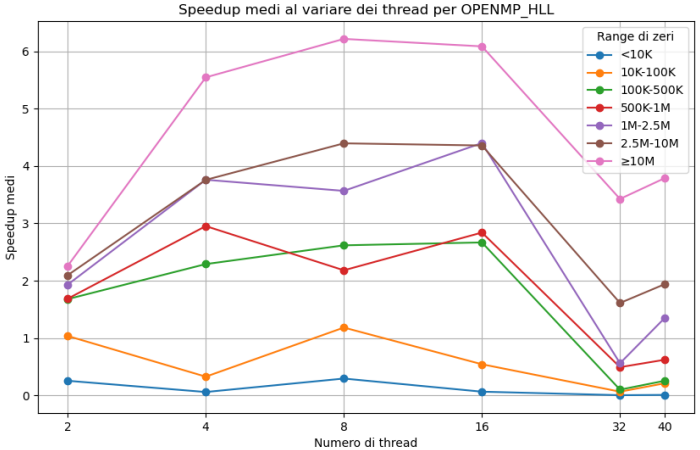
Risultati OpenMP HLL



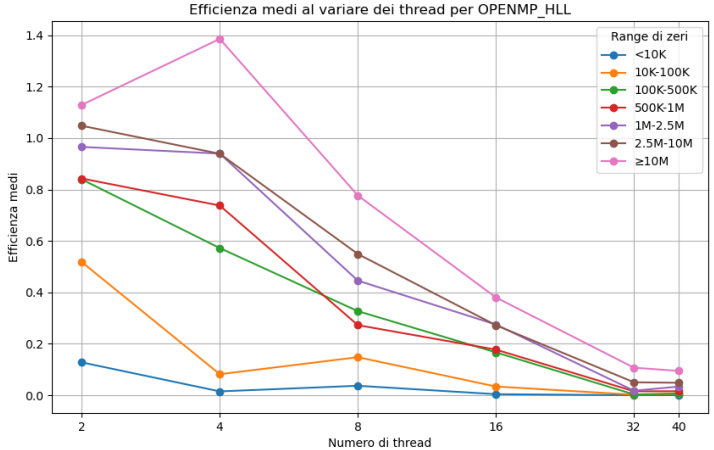
GFLOPS OpenMP HLL



Avg GFLOPS per Thread

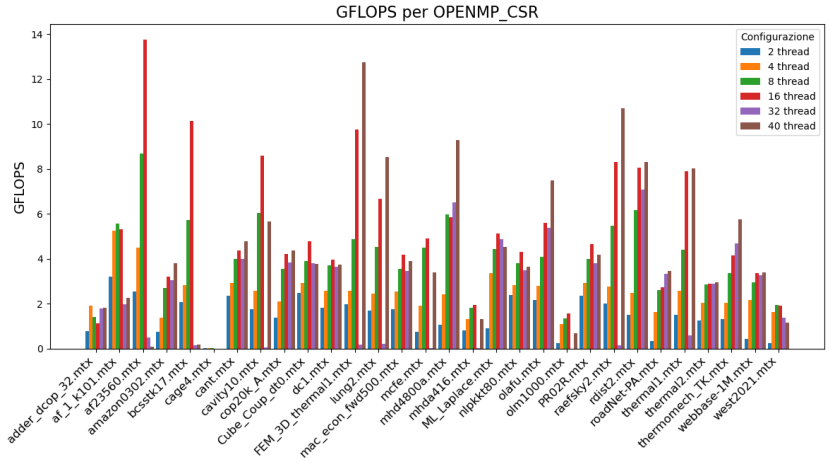


Avg Speedup

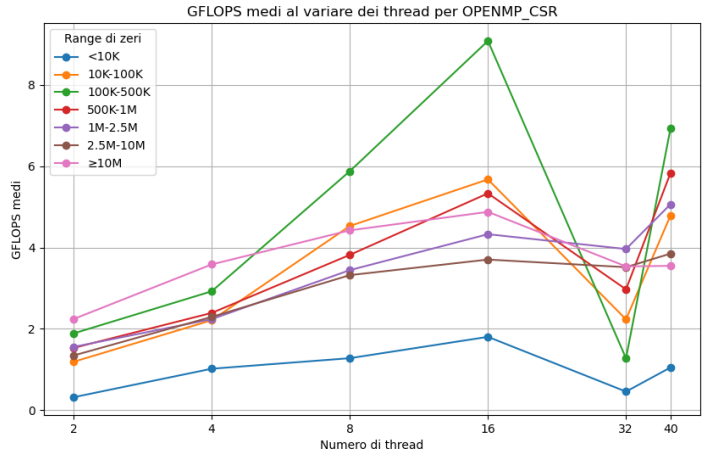


Avg Efficiency

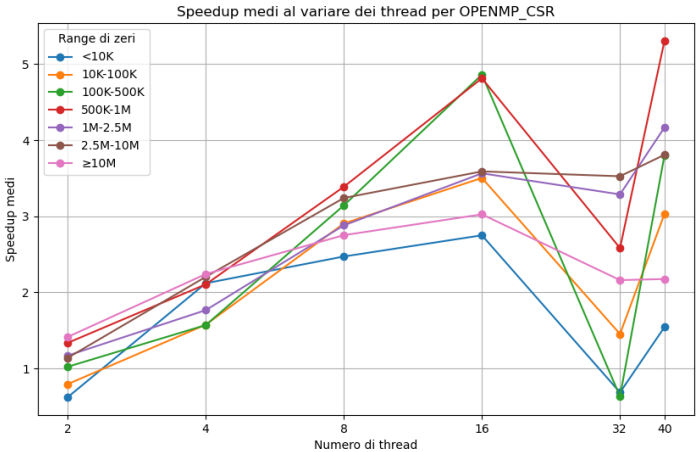
Risultati OpenMP CSR



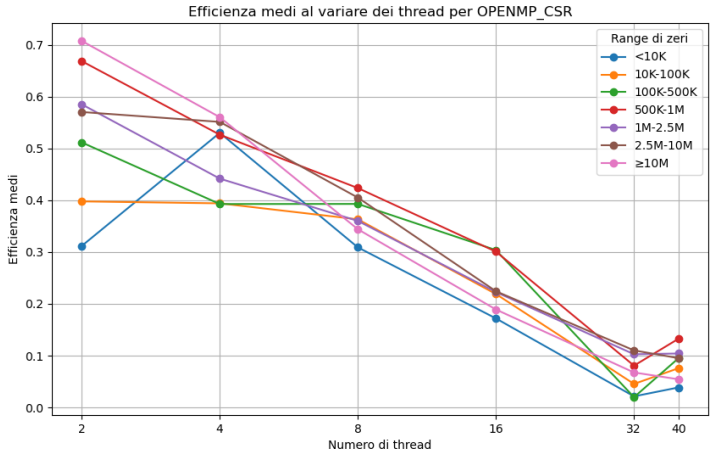
GFLOPS OpenMP CSR



Avg GFLOPS per Thread

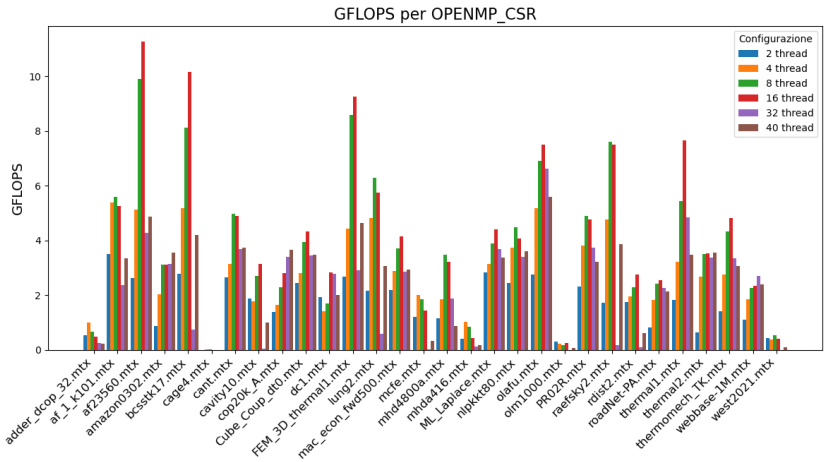


Avg Speedup

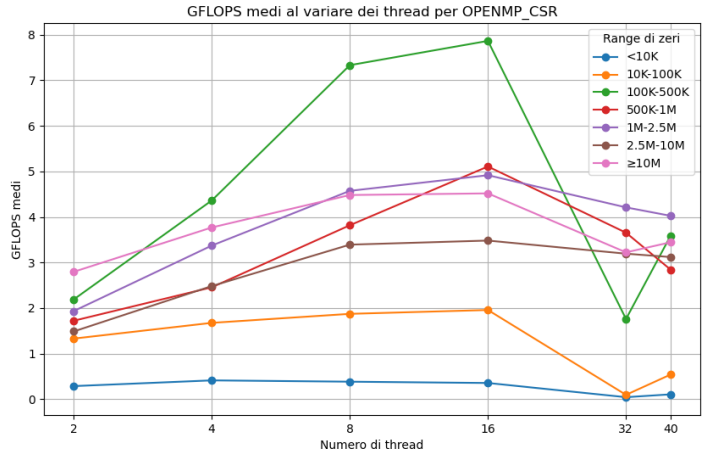


Avg Efficiency

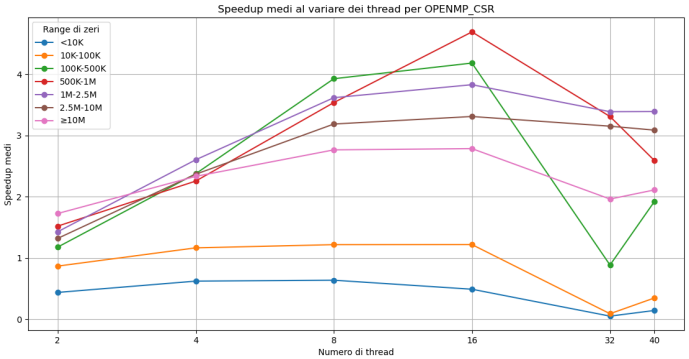
Risultati OpenMP CSR Guided



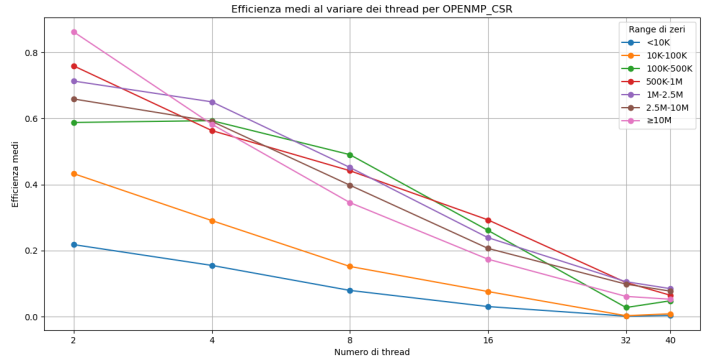
GFLOPS OpenMP CSR



Avg GFLOPS per Thread

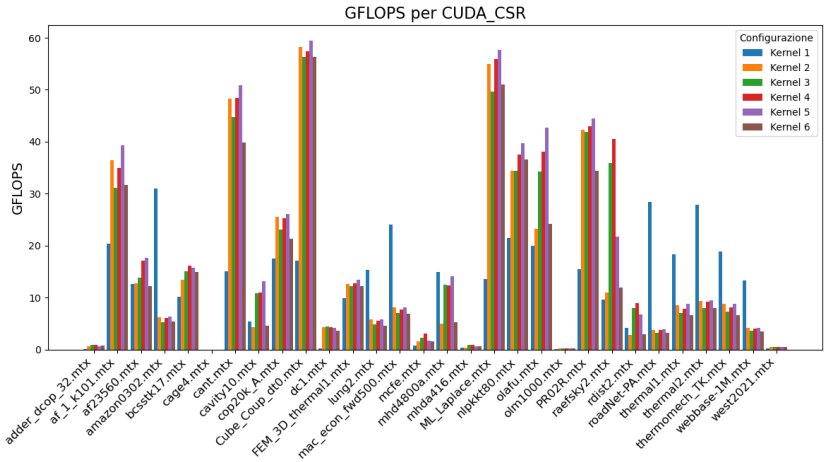


Avg Speedup

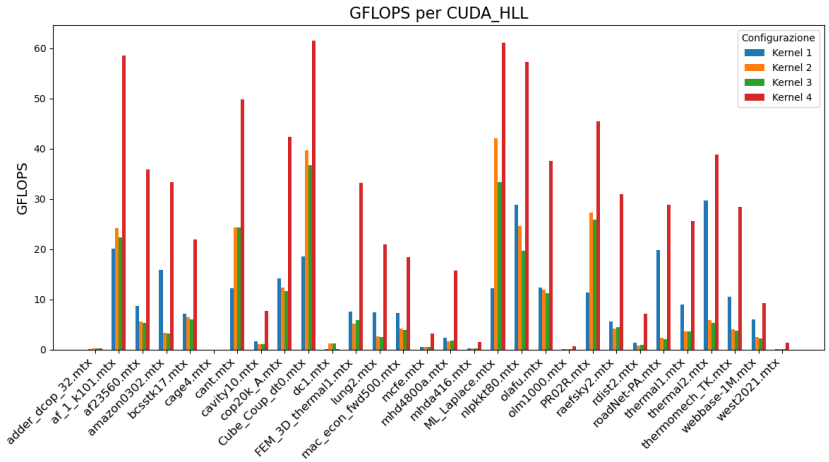


Avg Efficiency

Risultati CUDA

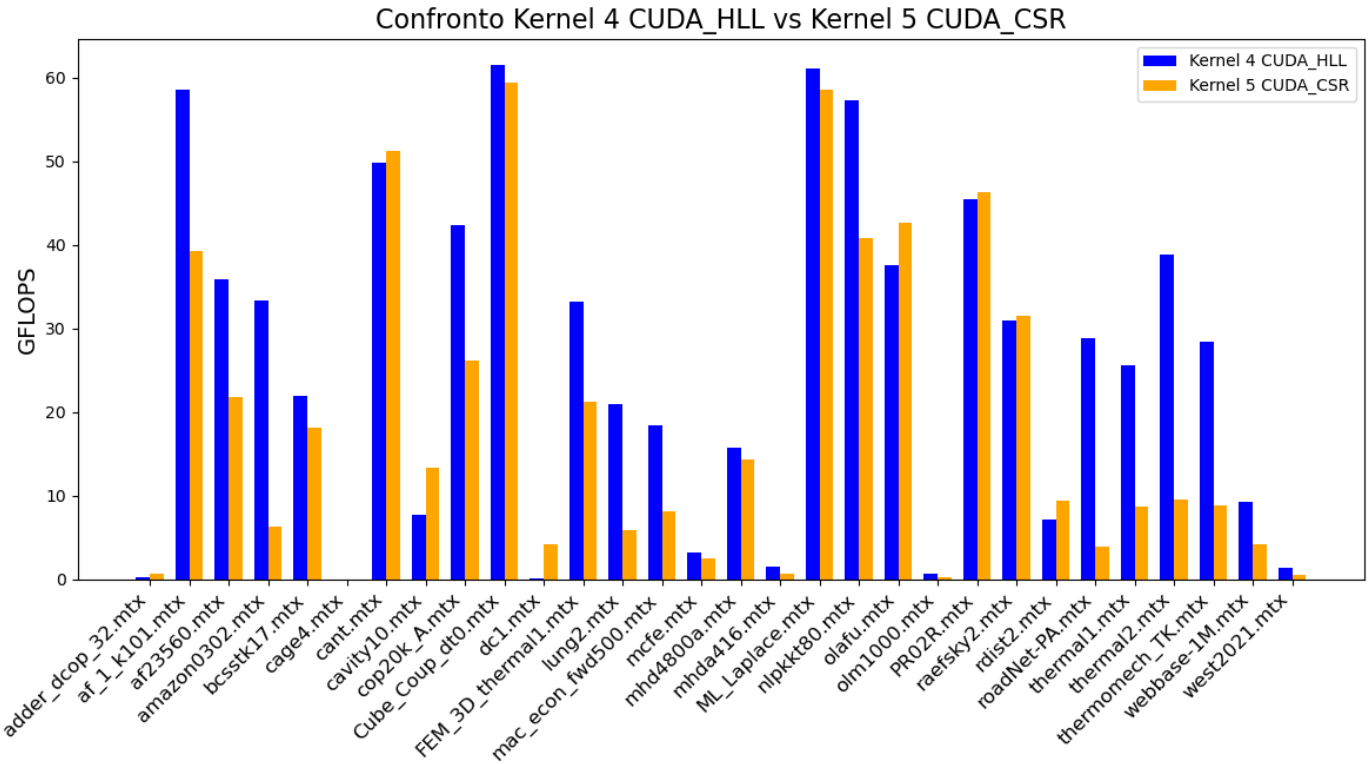


GFLOPS foreach CSR kernel



GFLOPS foreach HLL kernel

Risultati CUDA



Best kernel CSR vs Best kernel HLL

Grazie per l'attenzione!

Matteo Calzetta, Alessandro Lori