

Calcolo Parallelo per il Prodotto Matrice Sparsa-Vettore

Leonardo Pompili

Facoltà di Ingegneria Informatica

Università degli Studi di Roma Tor Vergata

Roma, Italia

leonardo.pompili@students.uniroma2.eu

Abstract—In questo documento viene descritto lo sviluppo e l’analisi di un nucleo di calcolo per il prodotto matrice sparsa-vettore (SpMV), un’operazione fondamentale in numerose applicazioni scientifiche e ingegneristiche. L’obiettivo è stato quello di implementare e valutare diverse strategie di parallelizzazione per l’operazione $y \leftarrow Ax$, dove A è una matrice sparsa. Sono stati considerati due formati di memorizzazione per la matrice A : Compressed Sparse Row (CSR) e Hybrid Linked List (HLL). Il parallelismo è stato implementato utilizzando i framework OpenMP per CPU multi-core e CUDA per GPU NVIDIA. Il lavoro comprende la fase di preprocessamento dei dati da file in formato MatrixMarket, la conversione nei formati CSR e HLL, l’implementazione delle versioni seriali e parallele dell’SpMV, e un’analisi dettagliata delle prestazioni in termini di GigaFLOPS, speedup ed efficienza, al variare dei parametri di parallelizzazione (numero di thread per OpenMP, dimensione dei blocchi per CUDA) e del parametro ‘HackSize’ per il formato HLL. I test sono stati condotti su un set di matrici di test standard della Suite Sparse Matrix Collection.

Index Terms—Prodotto Matrice-Vettore Sparso; SpMV; CSR; HLL; ELLPACK; OpenMP; CUDA.

I. INTRODUZIONE

Il prodotto tra una matrice sparsa ed un vettore (SpMV, Sparse Matrix-Vector Multiplication) è un’operazione computazionale critica in molti campi scientifici e ingegneristici, quali la risoluzione di sistemi di equazioni lineari, metodi iterativi, simulazioni fisiche, analisi di grafi e machine learning. La sparsità delle matrici, caratterizzata da una predominanza di elementi nulli, permette ottimizzazioni significative sia in termini di occupazione di memoria che di tempo di calcolo, rispetto alla gestione di matrici dense.

Nonostante la sua apparente semplicità matematica, l’implementazione efficiente di SpMV presenta diverse sfide. Gli accessi irregolari alla memoria, dovuti alla natura sparsa degli indici di colonna, e un basso rapporto tra operazioni aritmetiche e accessi alla memoria (bassa intensità computazionale) possono limitare le prestazioni, rendendo l’operazione spesso limitata dalla larghezza di banda della memoria (memory-bound).

Il parallelismo offre un percorso promettente per accelerare questo calcolo. Sfruttando le architetture moderne multi-core (CPU) e many-core (GPU), è possibile distribuire il carico di lavoro e ottenere significativi miglioramenti prestazionali. Il progetto presentato è volto all’esplorazione di due paradigmi di parallelismo comuni: OpenMP, per sistemi a memoria

condivisa su CPU, e CUDA, per l’elaborazione parallela su GPU NVIDIA.

Nei paragrafi che seguono vengono dettagliate le fasi di sviluppo: il preprocessamento dei dati, la rappresentazione dei formati di memorizzazione, le strategie di parallelizzazione per CPU e GPU, e un’analisi quantitativa delle prestazioni ottenute.

II. PREPROCESSAMENTO DEI DATI

La fase di preprocessing è un passaggio fondamentale per preparare i dati delle matrici ai successivi stadi di calcolo. Per il collaudo del nucleo SpMV è stata utilizzata una selezione di matrici dalla “Suite Sparse Matrix Collection”, dove le matrici sono state scaricate in formato MatrixMarket (.mtx). Il processo di caricamento e preparazione dei dati, encapsulato principalmente nella funzione `read_matrix_market_to_csr`, si articola in diverse fasi logiche.

A. Lettura dei File MatrixMarket

Il primo passo consiste nella lettura del file .mtx. Ogni file è strutturato con un *banner* iniziale che ne descrive le proprietà, seguito dalle dimensioni della matrice (numero di righe M , numero di colonne N e numero di elementi non-zero NZ memorizzati) e infine dalla lista di terne (riga, colonna, valore) che rappresentano gli elementi non-zero.

Il parser proposto analizza il banner per identificare tre proprietà chiave della matrice:

- **Formato:** Viene verificato che il formato sia coordinate, l’unico supportato per la lettura iniziale.
- **Tipo di dati:** Vengono gestiti i tipi `real`, `integer` e `pattern`. Nel caso di matrici *pattern*, dove i valori non sono esplicitamente memorizzati, viene implicitamente assegnato il valore 1.0 a ogni elemento non-zero.
- **Struttura:** Viene identificata la struttura della matrice, in particolare se è `general` o `symmetric`.

Durante la lettura, gli indici di riga e colonna, che nel formato MatrixMarket sono 1-based, vengono convertiti in 0-based per coerenza con la convenzione del linguaggio C.

B. Espansione delle Matrici Simmetriche

Una caratteristica comune delle matrici nella collezione è la loro simmetria. Per ottimizzare lo spazio su disco, i file .mtx per matrici simmetriche memorizzano solo gli elementi di un triangolo (solitamente quello inferiore, inclusa la diagonale). Tuttavia, per un corretto calcolo del prodotto matrice-vettore, è necessaria la rappresentazione completa della matrice.

La pipeline di preprocessing gestisce questa caratteristica in modo esplicito: se una matrice viene identificata come simmetrica, per ogni elemento non-zero (i, j) con valore v letto dal file, viene aggiunto anche il suo elemento simmetrico (j, i) con lo stesso valore v , a condizione che $i \neq j$. Questo processo di "espansione" garantisce che la struttura dati finale contenga tutti gli elementi non-zero della matrice completa e il conteggio totale degli NNZ viene aggiornato di conseguenza.

A valle di questa fase, la matrice, ora completa e rappresentata da una lista di terne, è pronta per essere convertita nei formati di memorizzazione ottimizzati (CSR e HLL) discussi nella sezione successiva.

III. RAPPRESENTAZIONE IN MEMORIA DELLE MATRICI SPARSE

La scelta del formato di memorizzazione è determinante per le performance dell'operazione SpMV. Un formato efficiente deve bilanciare due obiettivi: minimizzare l'occupazione di memoria, memorizzando solo gli elementi non-zero, e organizzare i dati in modo da favorire accessi regolari e prevedibili, massimizzando così la località dei dati e il throughput della memoria. Per questo progetto, sono stati implementati e analizzati due formati distinti.

A. Formato CSR (Compressed Sparse Row)

Il formato CSR è una delle rappresentazioni più diffuse e versatili per matrici sparse generiche. Una matrice A di dimensioni $M \times N$ con NZ elementi non-zero è rappresentata da tre array monodimensionali, come illustrato in Figura 1:

- **AS (Values):** Un array di tipo `float` di dimensione NZ , che memorizza i valori degli elementi non-zero della matrice, scanditi riga per riga, dalla prima all'ultima.
 - **JA (Column Indices):** Un array di tipo `int` di dimensione NZ , che contiene gli indici di colonna corrispondenti a ciascun valore nell'array AS.
 - **IRP (Row Pointers):** Un array di tipo `int` di dimensione $M + 1$. L'elemento `IRP [i]` funge da puntatore all'inizio dei dati della riga i negli array AS e JA. L'ultimo elemento, `IRP [M]`, memorizza il numero totale di non-zero, NZ . Di conseguenza, gli elementi della riga i si trovano negli indici compresi tra `IRP [i]` e `IRP [i+1] - 1`.

Il principale vantaggio del CSR risiede nella sua capacità di rappresentare qualsiasi struttura di sparsità e di permettere un’iterazione efficiente sulle righe, una caratteristica ben sfruttata negli algoritmi sequenziali e paralleli su CPU.

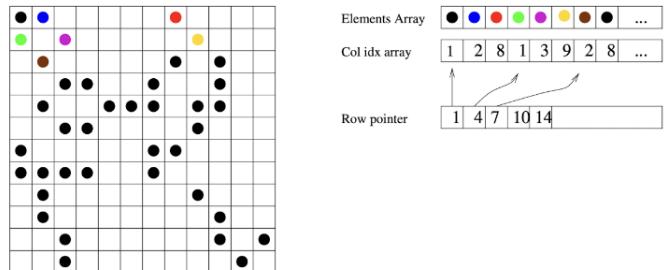


Fig. 1: Esempio di rappresentazione di una matrice sparsa in formato CSR.

B. Formato HLL (Hierarchical ELLPACK-like)

Mentre il CSR è molto flessibile, la variabilità del numero di elementi per riga può portare a un carico di lavoro sbilanciato e a pattern di accesso irregolari, problematici per le architetture SIMD/SIMT come le GPU. Il formato HLL è stato concepito per mitigare questi problemi introducendo una maggiore regolarità.

La strategia consiste nel partizionare orizzontalmente la matrice in blocchi contigui di `hack_size` righe. Ciascuno di questi blocchi viene poi memorizzato in un formato derivato da **ELLPACK**, come mostrato in Figura 2. Nel formato ELLPACK, se K è il numero massimo di elementi non-zero in una qualsiasi riga all'interno di un blocco, il blocco stesso (di R righe) viene memorizzato in due strutture dati rettangolari di dimensioni $R \times K$:

- **AS_ell (Values)**: Memorizza i valori dei coefficienti.
 - **JA_ell (Column Indices)**: Memorizza i corrispondenti indici di colonna.

Le righe del blocco che contengono meno di K elementi non-zero vengono riempite con valori di *padding*. Questo garantisce che ogni riga logica abbia la stessa lunghezza, regolarizzando la struttura dei loop di calcolo e facilitando la vettorizzazione e il parallelismo.

Per ottimizzare ulteriormente le performance su GPU, nell'implementazione proposta i dati di ogni blocco ELLPACK sono memorizzati in layout **column-major** (per colonne) anziché nel tradizionale row-major. Questa scelta garantisce che thread consecutivi di un warp, processando righe adiacenti, accedano a locazioni di memoria contigue, abilitando così l'accesso *coalescente* e massimizzando il throughput della memoria del device.

IV. IMPLEMENTAZIONE SPMV SERIALE

Prima di esplorare le strategie di parallelizzazione, è stata implementata una versione seriale del prodotto matrice-vettore. Questa implementazione funge da baseline di riferimento, essenziale per calcolare metriche fondamentali come lo speedup e l'efficienza delle controparti parallele.

Per la sua semplicità ed efficienza in un contesto sequenziale, è stato scelto il formato CSR come base per l'algoritmo seriale. L'implementazione, mostrata in Algoritmo 1, segue un approccio diretto:

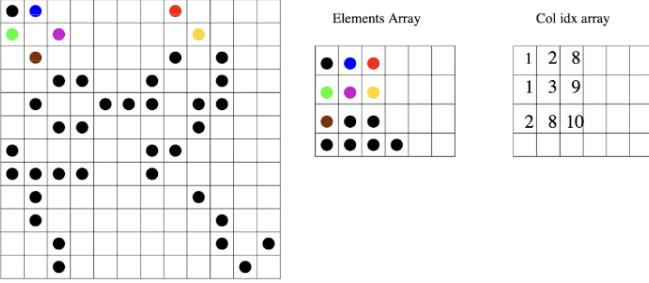


Fig. 2: Esempio di memorizzazione in formato HLL/ELL-PACK. La matrice viene riorganizzata in strutture dati rettangolari con padding.

- 1) Il ciclo principale itera su ciascuna riga i della matrice, da 0 a $M - 1$.
- 2) Per ogni riga, viene inizializzata una variabile di accumulazione, `sum`, a zero.
- 3) Utilizzando l'array dei puntatori `IRP`, si determinano gli indici di inizio e fine degli elementi non-zero della riga i all'interno degli array `JA` e `AS`.
- 4) Un ciclo interno itera su questi elementi, calcolando il prodotto tra il valore dell'elemento della matrice $A(i, j)$ e l'elemento corrispondente del vettore $x(j)$, accumulando il risultato in `sum`.
- 5) Al termine del ciclo interno, il valore finale di `sum` viene assegnato all'elemento $y(i)$ del vettore risultato.

Questo algoritmo rappresenta il punto di partenza standard con cui verranno confrontate tutte le ottimizzazioni successive.

Algorithm 1 SpMV Seriale in formato CSR

```

1: Input: Matrice  $A_{csr}$ , Vettore  $x$ 
2: Output: Vettore  $y$ 
3: procedure SERIALSPMVCSR( $A_{csr}, x, y$ )
4:   for  $i \leftarrow 0$  to  $M - 1$  do
5:      $sum \leftarrow 0.0$ 
6:     for  $k \leftarrow A_{csr}.\text{IRP}[i]$  to  $A_{csr}.\text{IRP}[i + 1] - 1$  do
7:        $col \leftarrow A_{csr}.\text{JA}[k]$ 
8:        $val \leftarrow A_{csr}.\text{AS}[k]$ 
9:        $sum \leftarrow sum + val \times x[col]$ 
10:     $y[i] \leftarrow sum$ 
```

V. OPENMP

La prima strategia di parallelizzazione adottata si basa su **OpenMP** (Open Multi-Processing), un'API standard per la programmazione parallela su architetture a memoria condivisa (CPU multi-core). Il modello di parallelismo di OpenMP, basato su direttive del compilatore, permette di parallelizzare sezioni di codice, tipicamente dei loop, in modo relativamente semplice.

Per ottenere un bilanciamento del carico efficace, si è optato per un **partizionamento statico manuale** del lavoro. A differenza dello scheduling dinamico, che può introdurre un

overhead non trascurabile, un partizionamento statico deciso a priori garantisce prevedibilità e riduce i costi di gestione.

A. Parallelizzazione con formato CSR

Per il formato CSR, l'unità di lavoro naturale da parallelizzare è la riga. Tuttavia, un semplice partizionamento che assegna a ogni thread un numero uguale di righe può risultare inefficiente se la distribuzione degli elementi non-zero (NNZ) è sbilanciata.

Per affrontare questo problema, è stata implementata una strategia di partizionamento statico basata sul **carico di lavoro effettivo (NNZ)**. Come mostrato in Algoritmo 2, il numero totale di non-zeri viene suddiviso equamente tra i thread, e una ricerca binaria sull'array `IRP` determina i confini delle righe (`row_bounds`) che corrispondono a tale suddivisione. Questa fase di preparazione viene eseguita una sola volta, fuori dal ciclo di misurazione del tempo.

Il kernel di calcolo (Algoritmo 3) riceve quindi questi limiti pre-calcolati, e ogni thread opera su un blocco di righe che, pur avendo dimensioni diverse, contiene un numero di non-zeri approssimativamente uguale, garantendo un buon bilanciamento del carico.

Algorithm 2 Preparazione dei limiti per CSR (su NNZ)

```

1: Input: Matrice  $A_{csr}$ ,  $num\_threads$ 
2: Output: Array row_bounds
3: procedure PREPARECSRBOUNDS( $A_{csr}, num\_threads$ )
4:    $nnz\_per\_thread \leftarrow \lceil A_{csr}.\text{nnz}/num\_threads \rceil$ 
5:    $row\_bounds[0] \leftarrow 0$ 
6:   for  $t \leftarrow 1$  to  $num\_threads - 1$  do
7:      $target\_nnz \leftarrow t \times nnz\_per\_thread$ 
8:      $row\_bounds[t] \leftarrow \text{BinarySearch}(A_{csr}.\text{IRP}, target\_nnz)$ 
9:    $row\_bounds[num\_threads] \leftarrow A_{csr}.\text{nrows}$ 
```

Algorithm 3 SpMV OpenMP per CSR con partizionamento su NNZ

```

1: Input: Matrice  $A_{csr}$ , Vettore  $x$ ,  $num\_threads$ , row_bounds
2: Output: Vettore  $y$ 
3: procedure OMPSPMVCSR( $A_{csr}, x, y, n, bounds$ )
  parallel region ( $num\_threads = n$ ):
4:    $tid \leftarrow \text{omp\_get\_thread\_num}()$ 
5:    $start \leftarrow bounds[tid]; end \leftarrow bounds[tid + 1]$ 
6:   for  $i \leftarrow start$  to  $end - 1$  do
7:     Calcola  $y[i]$  come in Algoritmo 1
```

B. Parallelizzazione con formato HLL

Per il formato HLL, la struttura stessa della matrice, partizionata in blocchi, suggerisce una strategia di parallelizzazione a granularità maggiore. Invece di parallelizzare sulle singole righe, il lavoro viene distribuito sui **blocchi HLL**.

Anche in questo caso, si adotta un partizionamento statico manuale (Algoritmo 4): a ogni thread viene assegnato un

sottoinsieme contiguo di blocchi da processare. Il kernel di calcolo (Algoritmo 5) fa iterare ogni thread sul proprio chunk di blocchi. All'interno di ogni blocco, il calcolo sulle righe viene eseguito sequenzialmente dal thread proprietario del blocco. Questo approccio riduce l'overhead di parallelizzazione e sfrutta la regolarità del formato ELLPACK.

Algorithm 4 Preparazione dei limiti per HLL

```

1: Input: Matrice  $A_{hll}$ ,  $num\_threads$ 
2: Output: Array  $block\_bounds$ 
3: procedure PREPAREHLLBOUNDS( $A_{hll}$ ,  $num\_threads$ )
4:    $num\_blocks \leftarrow A_{hll}.num\_blocks$ 
5:    $blocks\_per\_thread$  ←
6:    $\lceil num\_blocks / num\_threads \rceil$ 
7:   ...
8:   ▷ Calcolo statico dei limiti dei blocchi

```

Algorithm 5 SpMV OpenMP per HLL con partizionamento su blocchi

```

1: Input: Matrice  $A_{hll}$ , Vettore  $x$ ,  $num\_threads$ ,  

    $block\_bounds$ 
2: Output: Vettore  $y$ 
3: procedure OMPSPMVHLL( $A_{hll}, x, y, n, bounds$ )
   parallel region ( $num\_threads = n$ ):
4:    $tid \leftarrow omp\_get\_thread\_num()$ 
5:    $start\_b \leftarrow bounds[tid]; end\_b \leftarrow bounds[tid + 1]$ 
6:   for  $b \leftarrow start\_b$  to  $end\_b - 1$  do
7:     Processa tutte le righe del blocco  $b$ 

```

VI. CUDA

Per sfruttare il parallelismo massiccio offerto dalle moderne unità di elaborazione grafica (GPU), è stata sviluppata una seconda serie di implementazioni utilizzando il framework **CUDA** (Compute Unified Device Architecture) di NVIDIA. L'architettura di una GPU, composta da migliaia di core semplici organizzati in Streaming Multiprocessors (SM), è ideale per carichi di lavoro *data-parallel* come l'SpMV.

Tuttavia, ottenere performance elevate su GPU richiede un'attenzione particolare alla gestione della memoria e al pattern degli accessi.

A. Parallelizzazione con formato CSR

La strategia di parallelizzazione per il formato CSR su CUDA si basa sull'approccio "un thread per riga". Viene lanciata una griglia monodimensionale di thread, dove l'indice globale di ciascun thread viene mappato univocamente a una riga della matrice. Ogni thread è quindi responsabile del calcolo di un singolo elemento del vettore di output y .

Questa mappatura semplice, però, presenta un noto collo di bottiglia: gli accessi al vettore di input x sono indiretti e dipendono dalla struttura di sparsità della matrice, risultando in accessi alla memoria globale quasi certamente non-coalescenti. Per mitigare questa inefficienza, è stata impiegata la **texture memory**.

La texture memory è una cache hardware read-only sulla GPU, ottimizzata per la *località spaziale*. Quando un thread richiede un dato, la cache di texture carica non solo quell'elemento ma anche quelli adiacenti. Se i thread di uno stesso warp (un gruppo di 32 thread che eseguono la stessa istruzione) accedono a indirizzi di memoria vicini tra loro, anche se non perfettamente contigui, è molto probabile che le richieste successive alla prima trovino i dati già presenti in cache.

Nel kernel proposto (Algoritmo 6), il vettore x viene associato (bind) a un oggetto texture prima del lancio. All'interno del kernel, invece di un accesso diretto a ' $d_x[col_idx]$ ', si utilizza la funzione `tex1Dfetch`, che sfrutta questo meccanismo di caching. Questo approccio riduce significativamente la latenza media degli accessi al vettore x , portando a un notevole miglioramento delle performance rispetto a un'implementazione naïve.

Algorithm 6 Kernel CUDA per SpMV in CSR con Texture

```

1: Input: Matrice  $A_{csr}$  su GPU, Vettore  $d_y$ 
   ▷ Il vettore  $x$  è acceduto tramite la texture  $x\_tex$ 
2: Output: Vettore  $d_y$  riempito
3: procedure KERNELCUDACSR( $A_{csr}$ ,  $d_y$ )
4:    $row \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$ 
5:   if  $row < M$  then
6:      $sum \leftarrow 0.0$ 
7:      $start \leftarrow A_{csr}.\text{IRP}[row]$ 
8:      $end \leftarrow A_{csr}.\text{IRP}[row + 1]$ 
9:     for  $k \leftarrow start$  to  $end - 1$  do
10:        $col \leftarrow A_{csr}.\text{JA}[k]$ 
11:        $val \leftarrow A_{csr}.\text{AS}[k]$ 
12:        $x_{val} \leftarrow \text{tex1Dfetch}(x\_tex, col)$ 
13:        $sum \leftarrow sum + val \times x_{val}$ 
14:    $d_y[row] \leftarrow sum$ 

```

B. Parallelizzazione con formato HLL

L'implementazione CUDA mira a creare un pattern di accesso alla memoria più ideale possibile per l'architettura GPU. A tale scopo, è stata adottata una strategia di gestione dei dati relativamente sofisticata, basata su tre concetti fondamentali.

1) Data Flattening e Metadati: Anziché allocare memoria sulla GPU per ciascun blocco ELLPACK separatamente, i dati di tutti i blocchi (JA_ell e AS_ell) vengono concatenati e copiati in due singoli e grandi array continui sulla memoria del device (un processo noto come *data flattening*). Per permettere al kernel di localizzare i dati di uno specifico blocco all'interno di questi array "piatti", viene creata e copiata sulla GPU una piccola tabella di metadati. Ogni voce di questa tabella corrisponde a un blocco HLL e, invece di contenere puntatori, memorizza semplici **offset** (interi) che indicano la posizione di inizio dei dati di quel blocco. In questo modo, il kernel può calcolare l'indirizzo esatto dei dati con una sola operazione aritmetica, eliminando il pointer chasing.

2) *Accesso Coalescente e Kernel*: Il kernel per HLL, come per CSR, assegna un thread per riga globale. Tuttavia, grazie al layout *column-major* dei dati ELLPACK e alla struttura "flat", gli accessi agli array JA e AS diventano perfettamente **coalescenti**. Thread consecutivi (es. $t, t+1, t+2, \dots$) elaborano righe consecutive ($r, r+1, r+2, \dots$) e, ad ogni iterazione del loop interno, accedono a indirizzi di memoria adiacenti ('idx', 'idx+1', 'idx+2', ...). Questo schema permette alla GPU di servire le richieste di memoria di un intero warp con una singola transazione, massimizzando il throughput. Anche in questo caso, viene utilizzata la cache di texture per gli accessi al vettore x .

Algorithm 7 Kernel CUDA per SpMV in HLL Ottimizzato

```

1: Input: Metadati  $meta$ , Array  $JA_{flat}$ ,  $AS_{flat}$  su GPU
   ▷ Il vettore  $x$  è acceduto tramite la texture  $x\_tex$ 
2: Output: Vettore  $d\_y$  riempito
3: procedure                                     KERNELCUD-
   AHLL( $meta, JA_{flat}, AS_{flat}, d\_y$ )
4:    $i_{global} \leftarrow$  global thread index
5:   if  $i_{global} <$  total_rows then
6:      $b \leftarrow i_{global}/\text{block\_size}$ 
7:      $r_{local} \leftarrow i_{global}\%/\text{block\_size}$ 
8:      $block_{meta} \leftarrow meta[b]$ 
9:     if  $r_{local} < block_{meta}.\text{num\_rows}$  then
10:       $sum \leftarrow 0.0$ 
11:       $jabase \leftarrow block_{meta}.\text{ja\_start\_offset}$ 
12:       $asbase \leftarrow block_{meta}.\text{as\_start\_offset}$ 
13:      for  $k \leftarrow 0$  to  $block_{meta}.\text{max\_nz} - 1$  do
14:         $idx \leftarrow k \times block_{meta}.\text{num\_rows} + r_{local}$ 
15:         $val \leftarrow AS_{flat}[asbase + idx]$ 
16:        if  $val \neq 0.0$  then
17:           $col \leftarrow JA_{flat}[jabase + idx]$ 
18:           $sum \leftarrow sum + val \times$ 
             tex1Dfetch( $x\_tex, col$ )
19:           $d_y[i_{global}] \leftarrow sum$ 

```

VII. MISURAZIONE DELLE PRESTAZIONI E RISULTATI

La valutazione delle performance delle diverse implementazioni condotta è volta a garantire la riproducibilità e l'affidabilità delle misure. In questa sezione vengono riportate le metriche utilizzate e un'analisi dettagliata dei risultati ottenuti.

A. Metodologia di Misurazione e Metriche

Per ogni combinazione di matrice, formato e configurazione parallela, il nucleo di calcolo è stato eseguito per NUM_RUNS (pari a 10) volte. Il tempo di esecuzione medio, T_{avg} , è stato poi utilizzato per calcolare le metriche di performance. Questa media permette di mitigare fluttuazioni dovute a fattori esterni del sistema operativo.

La misurazione del tempo è stata effettuata come segue:

- **Seriiale:** Tramite la funzione `clock()` della libreria C standard.

- **OpenMP:** Tramite la funzione `omp_get_wtime()`, che restituisce il wall-clock time con alta precisione.
- **CUDA:** Tramite gli eventi CUDA (`cudaEventCreate`, `cudaEventRecord`, `cudaEventSynchronize`). Questo approccio misura esclusivamente il tempo di esecuzione del kernel sulla GPU, escludendo i tempi di trasferimento dati Host-Device.

Le metriche principali utilizzate per l'analisi sono:

- 1) **GFLOPS (Giga Floating Point Operations Per Second):** Misura la potenza di calcolo effettiva. Per l'SpMV, ogni elemento non-zero (NNZ) richiede una moltiplicazione e un'addizione (2 FLOPs).

$$\text{GFLOPS} = \frac{2 \times \text{NNZ}}{T_{avg} \times 10^9} \quad (1)$$

- 2) **Speedup:** Misura il miglioramento prestazionale rispetto all'implementazione seriale di riferimento (basata su CSR).

$$\text{Speedup} = \frac{T_{serial}}{T_{parallel}} \quad (2)$$

- 3) **Efficienza:** Indica quanto bene vengono sfruttati i processori. Un'efficienza pari a 1 (o 100%) rappresenta uno scaling lineare ideale.

$$\text{Efficienza} = \frac{\text{Speedup}}{\text{Numero di Thread}} \quad (3)$$

B. Risultati OpenMP

L'analisi delle performance per le implementazioni OpenMP si è concentrata sulla valutazione della scalabilità al variare del numero di thread (da 1 a 40). L'adozione di un partizionamento statico del carico, basato sul numero di non-zeri per il formato CSR e sul numero di blocchi per il formato HLL, si è rivelata una strategia efficace per ottenere risultati stabili.

- 1) **Analisi della Scalabilità:** Le performance e lo speedup delle implementazioni CSR e HLL al variare del numero di thread sono illustrate nelle Figure 3, 4, 5 e 6.

Entrambi i formati mostrano un comportamento di scaling positivo per quasi tutte le matrici: all'aumentare del numero di thread, i GFLOPS crescono. Come atteso, questo scaling non è infinito. Per molte matrici, specialmente quelle con un alto numero di non-zeri (es. `cant.mtx`, `ML_Laplace.mtx`), si osserva un picco di performance intorno ai 20 thread. Per le matrici più piccole o con meno non-zeri (es. `cop20k_A.mtx`), il picco di performance si raggiunge con un numero inferiore di thread. In questi casi, l'overhead introdotto dall'aggiungere nuovi thread supera rapidamente i benefici della parallelizzazione.

I grafici dell'efficienza (Figure 7 e 8) rivelano un'ampia gamma di comportamenti che dipendono strettamente dalle caratteristiche delle singole matrici.

La tendenza generale è una diminuzione dell'efficienza all'aumentare dei thread, causato dall'overhead di parallelizzazione che diventa progressivamente più rilevante. Tuttavia, si osservano due comportamenti estremi.

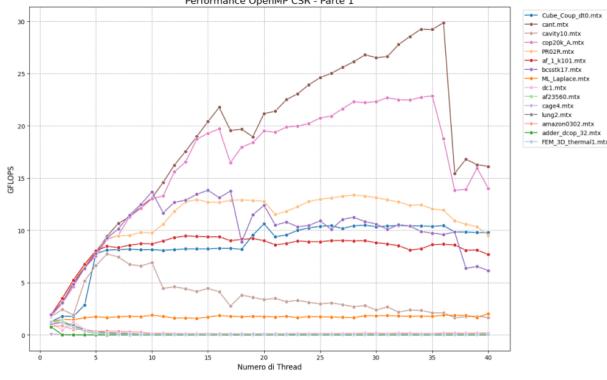
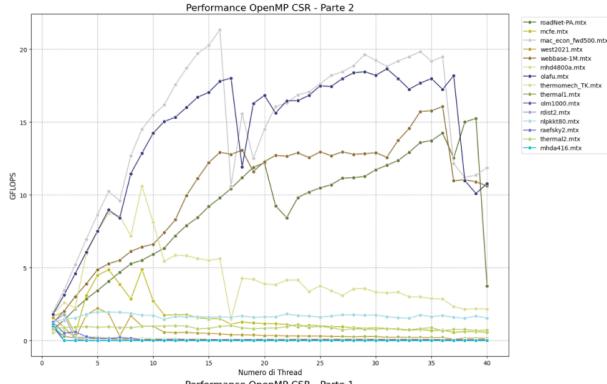


Fig. 3: Performance (GFLOPS) per SpMV OpenMP in formato CSR.

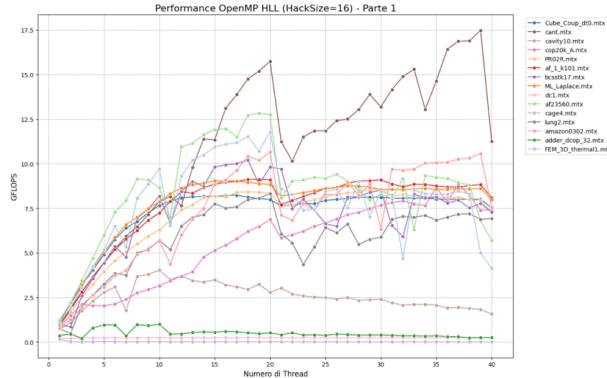
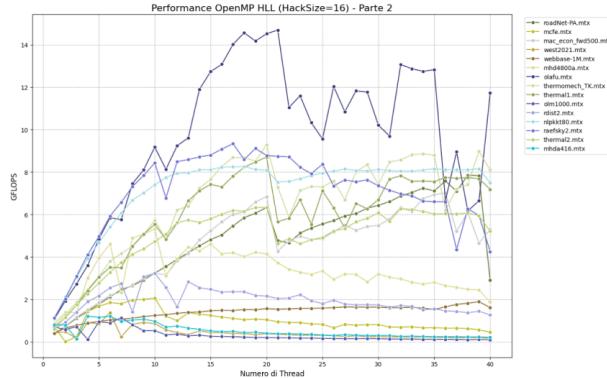


Fig. 4: Performance (GFLOPS) per SpMV OpenMP in formato HLL (HackSize=16).

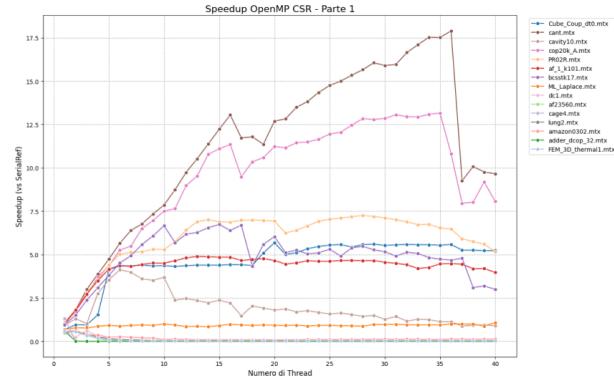
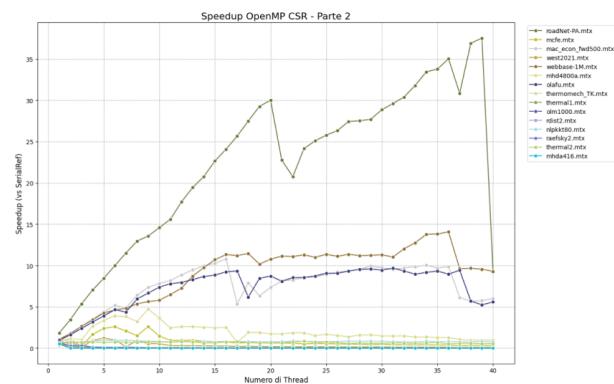


Fig. 5: Speedup per SpMV OpenMP in formato CSR.

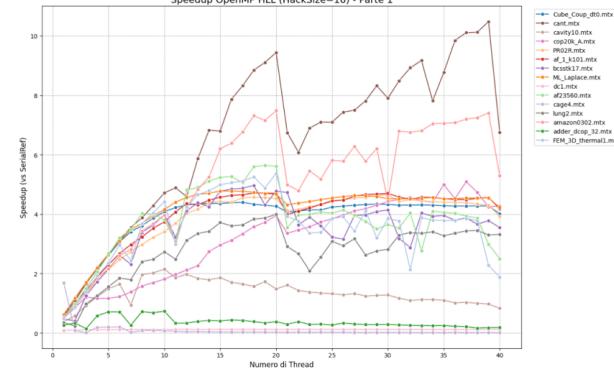
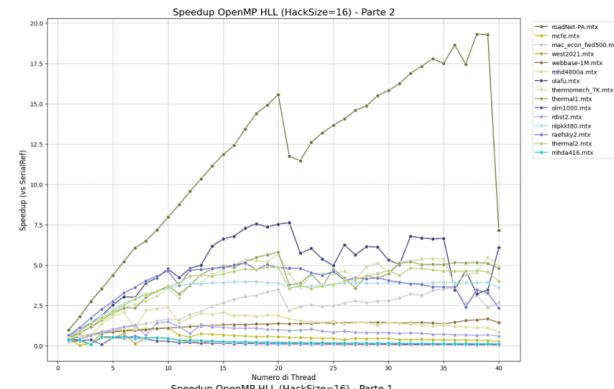


Fig. 6: Speedup SpMV OpenMP in formato HLL (HackSize=16).

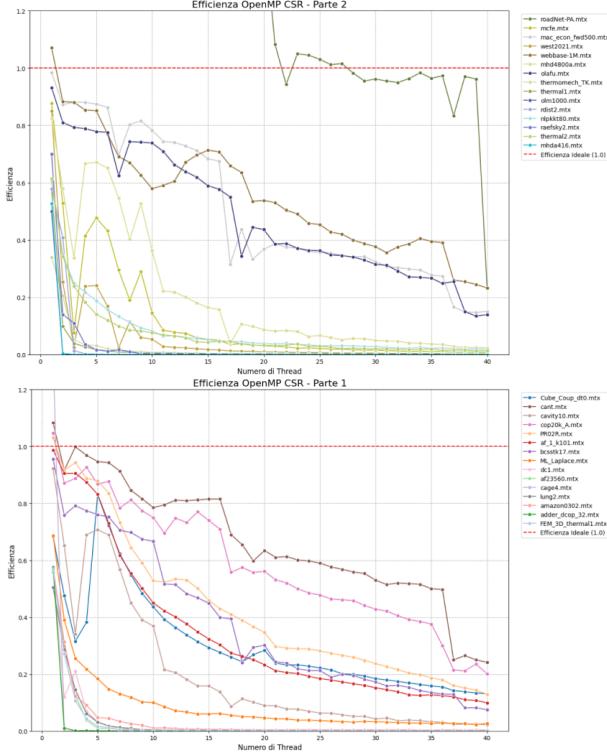


Fig. 7: Efficienza per SpMV OpenMP in formato CSR.

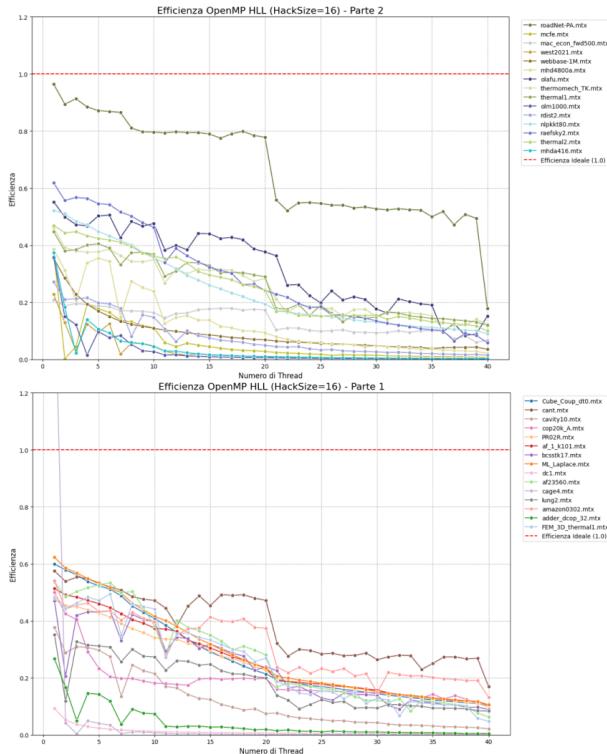


Fig. 8: Efficienza per SpMV OpenMP in formato HLL (HackSize=16).

Da un lato, matrici di grandi dimensioni e con una forte località spaziale, come roadNet-PA (un grafo di una rete stradale), esibiscono un'efficienza iniziale notevolmente superiore a 1.0. Questo fenomeno si verifica perché la versione seriale è pesantemente limitata dalla latenza della memoria RAM, essendo costretta a frequenti *cache miss* per accedere a un vettore di input molto grande. Quando il problema viene suddiviso tra più core, la porzione di dati gestita da ciascun thread diventa sufficientemente piccola.

Dall'altro lato, si osserva il comportamento della matrice cage4. Essendo estremamente piccola (9x9 righe con 49 NNZ), il tempo di calcolo seriale è trascurabile. L'altissima efficienza iniziale seguita da un crollo verticale è un artefatto di misurazione su un problema troppo piccolo per essere parallelizzato efficacemente. L'overhead introdotto dalla creazione e sincronizzazione anche solo di due thread supera di gran lunga il tempo risparmiato, rendendo il parallelismo controproducente. Questo caso limite dimostra che esiste una soglia di complessità del problema al di sotto della quale la parallelizzazione non è vantaggiosa.

2) *Confronto tra Formati: CSR vs HLL:* Nei grafici a barre (Figure 9, 10 e 11) è mostrato un confronto diretto sulle diverse metriche tra i due formati.

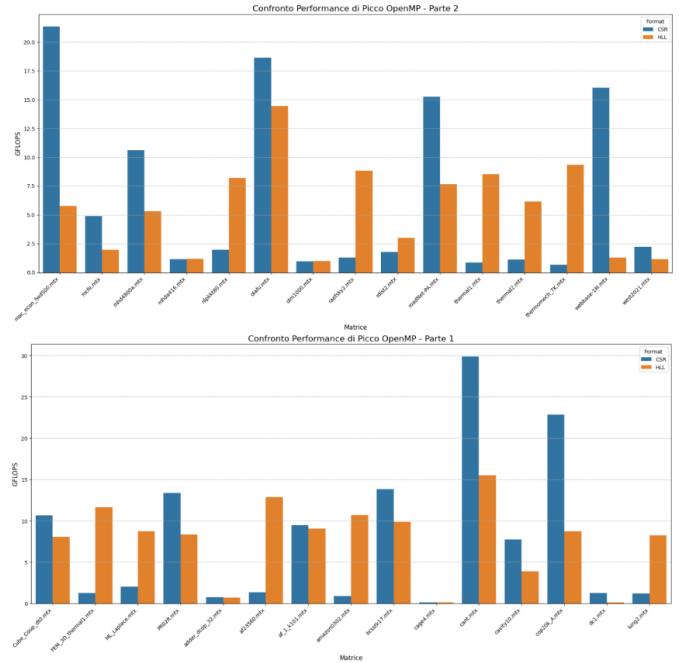


Fig. 9: Confronto delle performance di picco (GFLOPS) tra CSR e HLL in OpenMP.

Per la maggior parte delle matrici testate, l'implementazione CSR offre performance superiori a quella HLL in ambiente OpenMP.

La strategia di partizionamento per CSR, basata sul bilanciamento degli NNZ, si è dimostrata estremamente efficace nel distribuire equamente il lavoro effettivo. Questo permette a tutti i thread di operare con un carico simile, massimizzando l'utilizzo dei core.

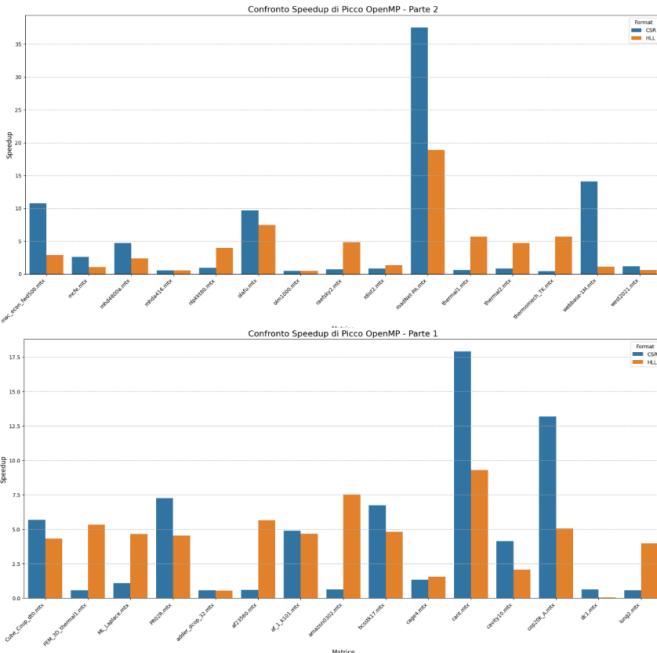


Fig. 10: Confronto su speedup tra CSR e HLL in OpenMP.

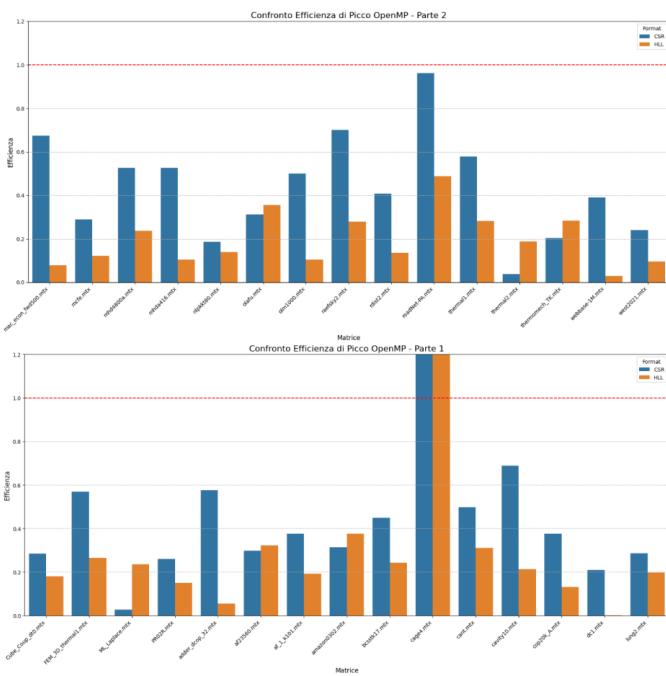


Fig. 11: Confronto sull'efficienza tra CSR e HLL in OpenMP.

Il formato HLL, pur regolarizzando la struttura, introduce un overhead. Il calcolo per accedere agli elementi (layout column-major) è leggermente più complesso, e il *padding* inserito per rendere i blocchi rettangolari introduce operazioni inutili su elementi zero. Su CPU, dove il branching non è troppo costoso, l'if per saltare i valori nulli è efficiente, ma il lavoro sprecato nel leggere dati di padding rimane.

Il formato HLL mostra performance competitive o superiori solo in rari casi, tipicamente per matrici con una struttura molto regolare dove l'overhead di padding è minimo.

3) *Analisi del formato HLL*: Per il formato HLL, la parallelizzazione è stata implementata tramite un partizionamento statico sui blocchi della matrice. Una variabile chiave di questo formato è il parametro *HackSize*, che definisce il numero di righe per ciascun blocco ELLPACK. Sono stati condotti test per diversi valori di *HackSize* (16, 32 e 64) al fine di valutarne l'impatto sulle performance.

I risultati, illustrati a titolo di esempio per *HackSize*=32 e *HackSize*=64 nelle Figure 12, 13, 14, 15, 16, e 17, mostrano un comportamento di scaling generalmente simile a quello del CSR, con un picco di performance che si manifesta intorno ai 20 thread.

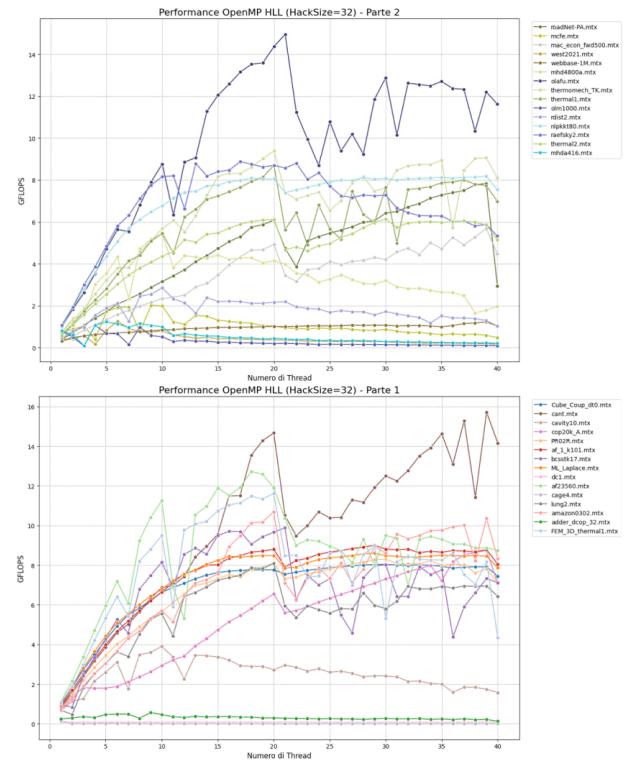


Fig. 12: Performance (GFLOPS) per SpMV OpenMP in formato HLL con HackSize=32.

La scelta di questo parametro influenza direttamente il trade-off tra la regolarità dei blocchi e l'overhead di padding.

- Un **HackSize piccolo** (es. 16) crea molti blocchi piccoli. Questo può portare a un buon bilanciamento del carico tra i thread, poiché ogni thread gestisce un numero maggiore di blocchi più piccoli. Tuttavia, se la

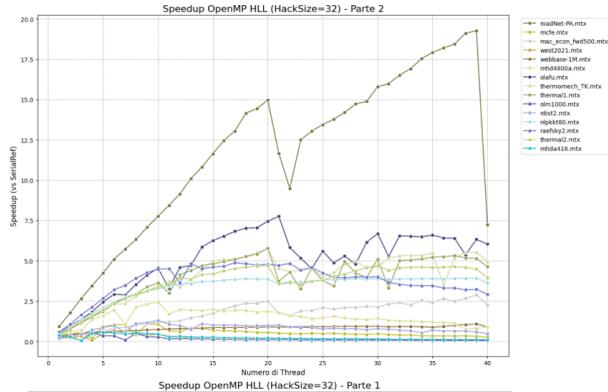


Fig. 13: Speedup per SpMV OpenMP in formato HLL con HackSize=32.

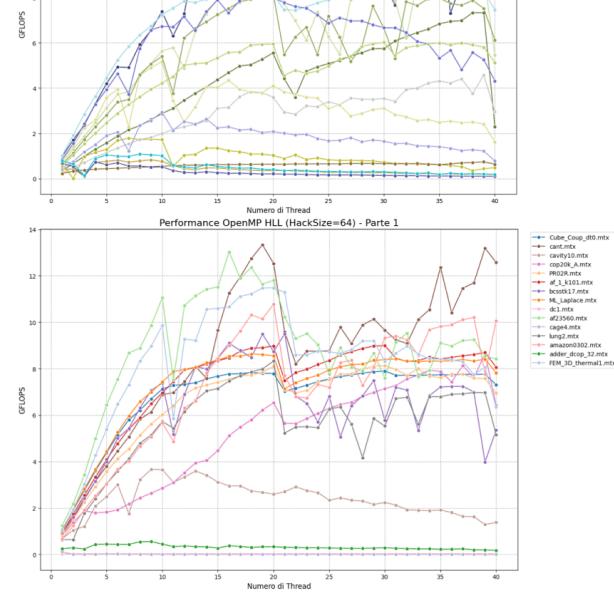
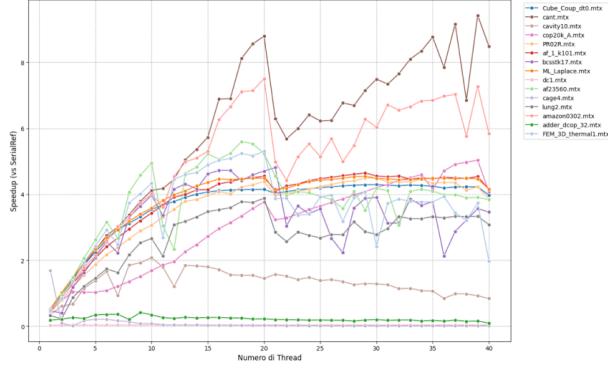


Fig. 15: Performance (GFLOPS) per SpMV OpenMP in formato HLL con HackSize=64.

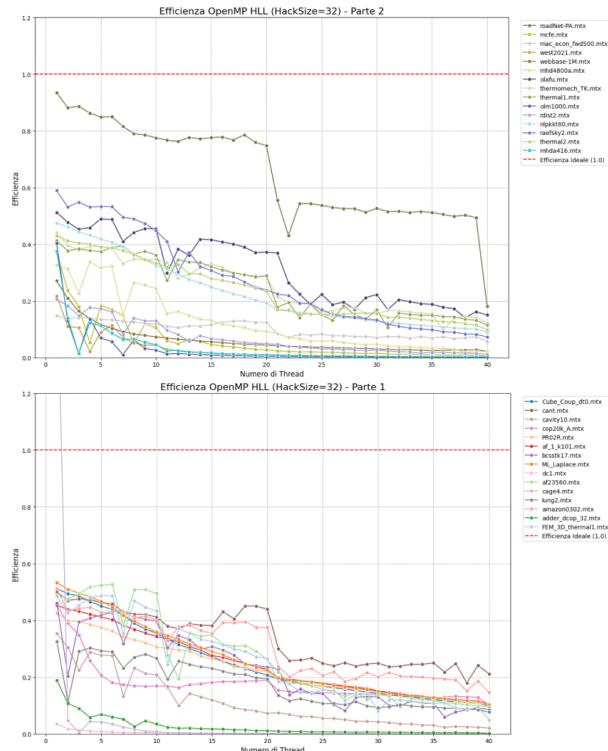


Fig. 14: Efficienza per SpMV OpenMP in formato HLL con HackSize=32.

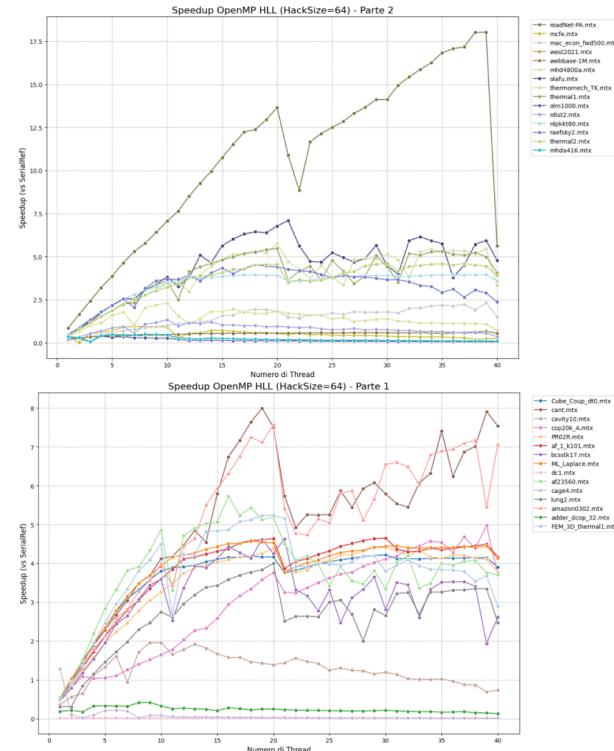


Fig. 16: Speedup per SpMV OpenMP in formato HLL con HackSize=64.

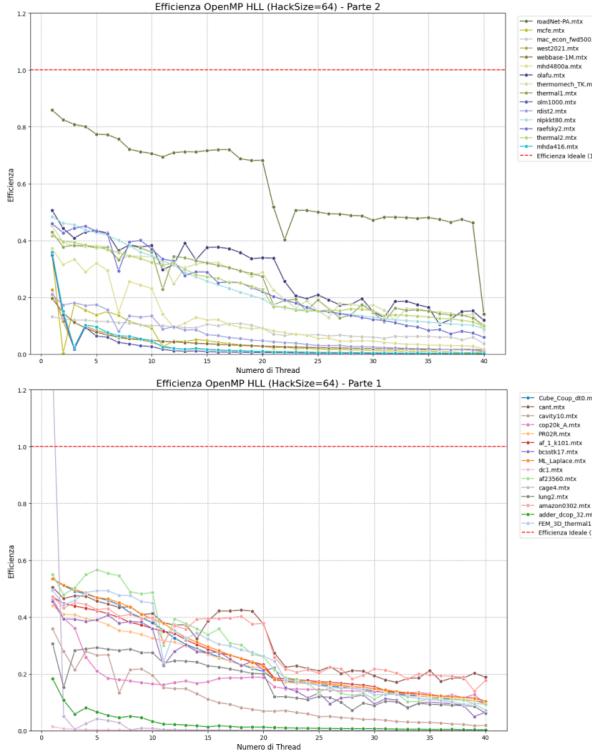


Fig. 17: Efficienza per SpMV OpenMP in formato HLL con HackSize=64.

distribuzione degli NNZ è molto irregolare, la probabilità di avere un'elevata varianza nella lunghezza delle righe all'interno di un blocco piccolo aumenta, portando a un overhead di padding significativo.

- Un **HackSize grande** (es. 64) crea meno blocchi, ma più grandi. Questo può ridurre l'overhead di scheduling, ma aumenta il rischio di sbilanciamento del carico se alcuni blocchi sono computazionalmente molto più "pesanti" di altri. D'altra parte, su un blocco più grande, la lunghezza massima delle righe (`max_nz_per_row`) tende a stabilizzarsi, potenzialmente riducendo l'overhead di padding relativo per le righe più dense.

Dai risultati sperimentali emerge che non esiste un valore di `HackSize` universalmente ottimale. La scelta migliore dipende strettamente dalla struttura della matrice. Ad esempio, per matrici con una distribuzione degli NNZ relativamente uniforme, un `HackSize` più grande potrebbe essere vantaggioso. Per matrici con "burst" di righe dense, un `HackSize` più piccolo potrebbe adattarsi meglio, isolando queste regioni irregolari in blocchi più piccoli.

C. Risultati CUDA

Le implementazioni su GPU sono state valutate facendo variare la dimensione del blocco di thread tra i valori {128, 256, 512, 1024}, al fine di identificare la configurazione ottimale per massimizzare l'occupazione dei multiprocessori (SM) della GPU.

1) *Analisi del formato CSR:* I risultati per l'implementazione CSR, che sfrutta la cache di texture per mitigare gli accessi non coalescenti al vettore x , sono presentati nelle Figure 18 e 19.

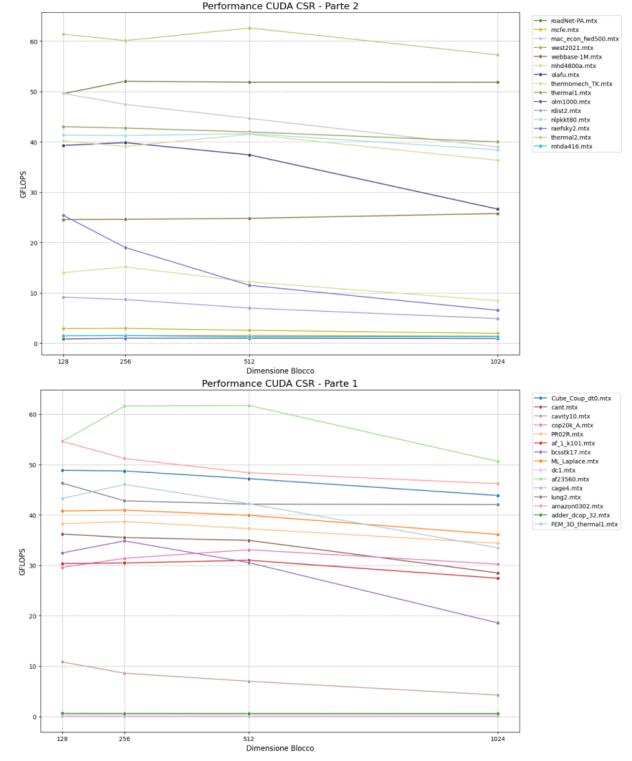


Fig. 18: Performance (GFLOPS) per SpMV CUDA in formato CSR al variare della dimensione del blocco.

Le performance mostrano una moderata dipendenza dalla dimensione del blocco. Per la maggior parte delle matrici, si osserva un punto di ottimo con blocchi di 256 o 512 thread. Blocchi più piccoli (128) potrebbero non essere sufficienti a nascondere la latenza della memoria, mentre blocchi più grandi (1024) possono portare a una riduzione delle performance a causa di una minore flessibilità nello scheduling dei blocchi sui multiprocessori. L'uso della texture memory si rivela efficace, consentendo di ottenere speedup significativi rispetto alla versione seriale, sebbene le performance assolute rimangano limitate dalla natura intrinsecamente irregolare del formato.

2) *Analisi del formato HLL:* L'implementazione HLL, basata su *data flattening* e layout *column-major*, è stata testata con diversi valori di `HackSize`. I risultati, illustrati nelle Figure 20 e 21 per `HackSize`=32, e 22 e 23 per `HackSize`=64, dimostrano l'efficacia di questa strategia.

L'accesso coalescente ai dati della matrice, reso possibile dal layout *column-major*, permette di sfruttare quasi appieno la larghezza di banda della memoria della GPU. Questo si traduce in performance e speedup nettamente superiori a quelli del formato CSR per la maggior parte delle matrici. Lo speedup per matrici come `roadNet-PA.mtx` supera il valore di 100x, un risultato notevole che evidenzia come l'allineamento della struttura dati all'architettura hardware sia fondamentale per il

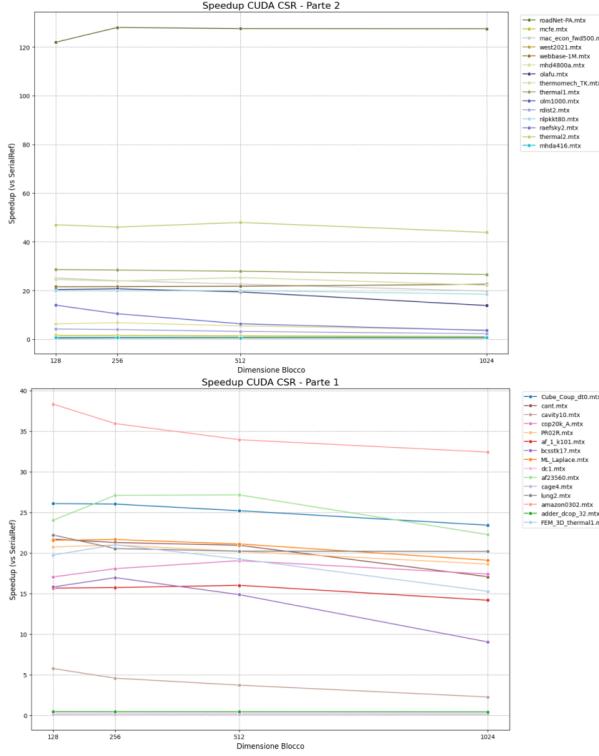


Fig. 19: Speedup per SpMV CUDA in formato CSR.

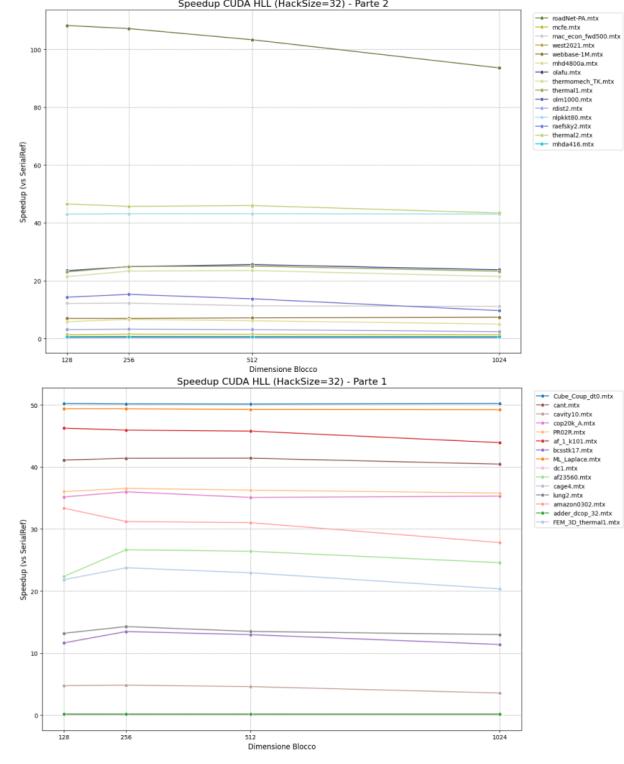


Fig. 21: Speedup per SpMV CUDA in formato HLL (HackSize=32).

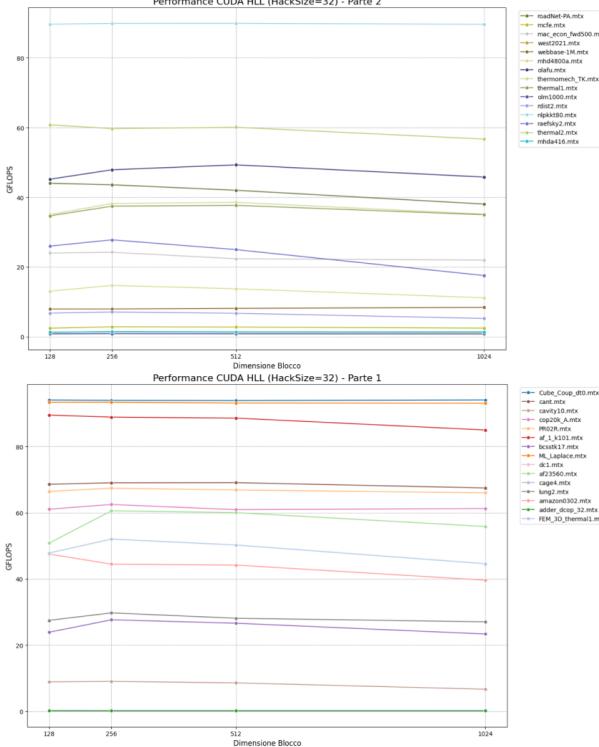


Fig. 20: Performance (GFLOPS) per SpMV CUDA in formato HLL (HackSize=32) al variare della dimensione del blocco.

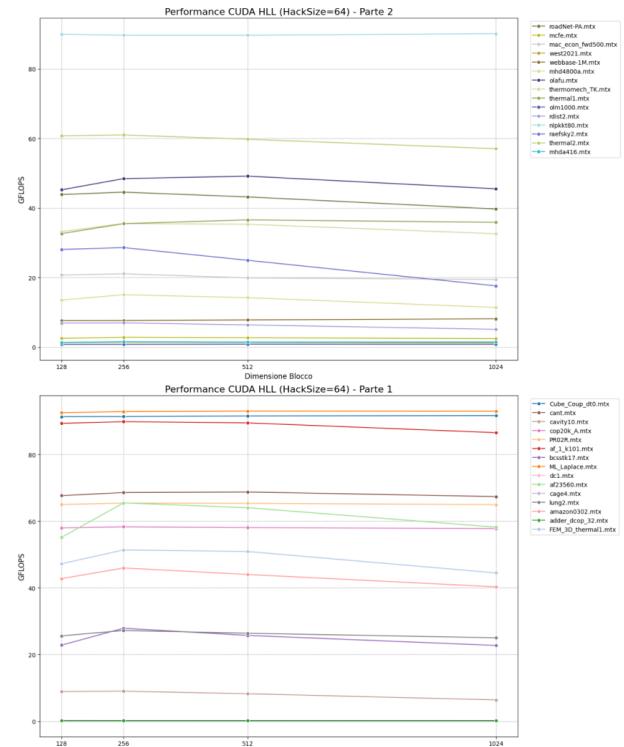


Fig. 22: Performance (GFLOPS) per SpMV CUDA in formato HLL (HackSize=64) al variare della dimensione del blocco.

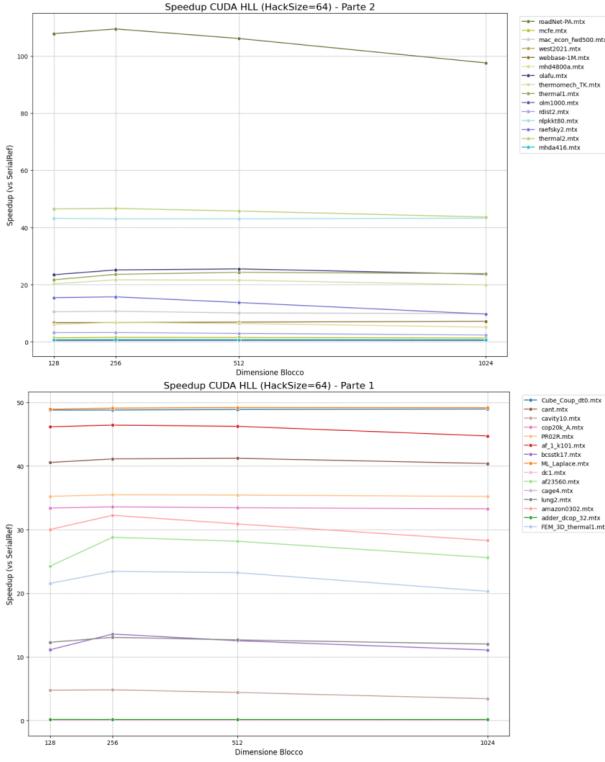


Fig. 23: Speedup per SpMV CUDA in formato HLL (HackSize=64).

calcolo ad alte prestazioni. Similmente al CSR, la dimensione del blocco ottimale si attesta spesso tra 256 e 512.

D. Confronto Generale e Discussione

Il confronto delle performance di picco ottenute da ciascuna implementazione permette di trarre conclusioni definitive sull'efficacia delle diverse strategie.

Come illustrato nelle Figure 24 e 25, emerge un quadro chiaro: su architettura GPU, il formato HLL ottimizzato domina nettamente il formato CSR. Per quasi tutte le matrici, e in particolare per quelle di grandi dimensioni, l'implementazione HLL raggiunge GFLOPS e speedup significativamente più elevati. Questo risultato conferma l'ipotesi che su un'architettura SIMD (Single Instruction, Multiple Thread) come quella di una GPU, la regolarizzazione degli accessi alla memoria tramite un formato come ELLPACK con layout column-major è la strategia vincente. L'overhead del padding introdotto da HLL è un costo trascurabile rispetto all'enorme guadagno ottenuto dagli accessi coalescenti.

Il formato CSR, pur essendo ottimizzato con la cache di texture, rimane intrinsecamente limitato dalla sua struttura, che non si adatta bene al modello di esecuzione parallela della GPU. Vince il confronto solo in rari casi, tipicamente su matrici con una struttura tale per cui l'overhead di padding di HLL diventa particolarmente sfavorevole.

1) *Impatto del Parametro HackSize*: L'analisi dei diversi valori di HackSize (16, 32, 64) per il formato HLL ha

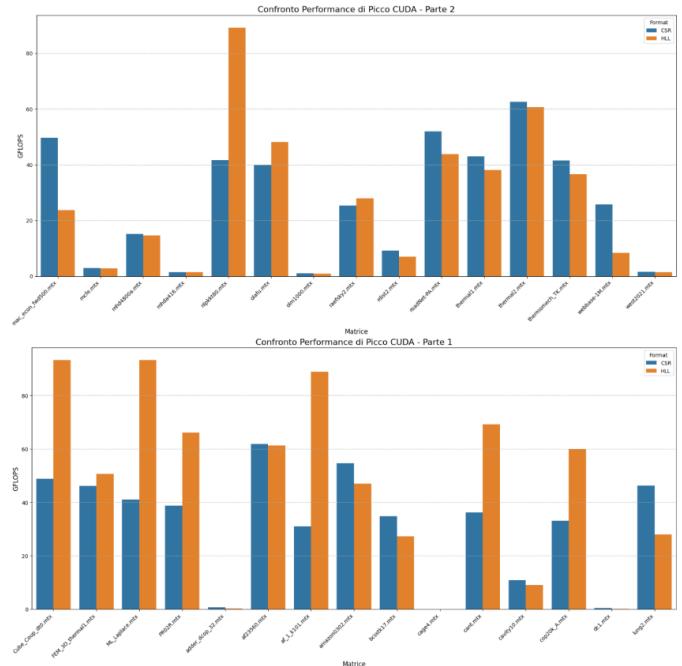


Fig. 24: Confronto delle performance di picco (GFLOPS) tra CSR e HLL in CUDA.

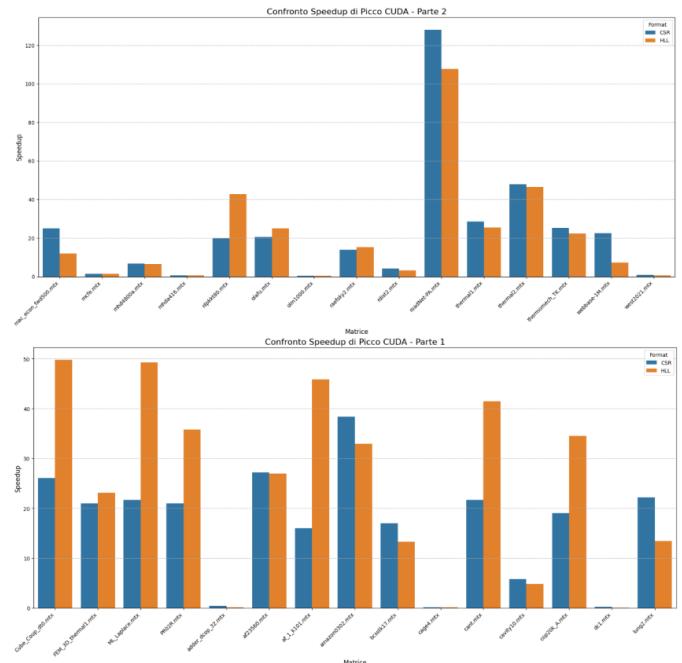


Fig. 25: Confronto dello speedup tra CSR e HLL in CUDA.

mostrato che non esiste un valore universalmente ottimale. La scelta migliore dipende dalla struttura interna della matrice.

- Un **HackSize piccolo** (es. 16) tende a performare bene su matrici con una distribuzione degli NNZ molto irregolare, poiché permette di isolare meglio le regioni dense.
- Un **HackSize più grande** (es. 64) può essere vantaggioso per matrici più uniformi, riducendo l'overhead relativo dei metadati.

Tuttavia, a differenza dell'ambiente OpenMP, su CUDA le differenze di performance tra i vari HackSize sono meno pronunciate. Questo suggerisce che, una volta garantito l'accesso coalescente, il beneficio principale è già stato ottenuto, e l'ulteriore tuning del parametro HackSize porta a miglioramenti marginali. Per semplicità e robustezza, un valore intermedio come 32 si è dimostrato una scelta solida per un'ampia gamma di matrici.