

Calcolo Parallelo per il Prodotto tra una Matrice Sparsa e un Vettore

Sara Malaspina

Facoltà di Ingegneria Informatica

Università degli Studi di Roma Tor Vergata
sara.malaspina@alumni.uniroma2.eu

Silvia Perelli

Facoltà di Ingegneria Informatica

Università degli Studi di Roma Tor Vergata
silvia.perelli@alumni.uniroma2.eu

Sommario—L’obiettivo di questo lavoro è quello di sviluppare un nucleo di calcolo per il prodotto tra una matrice sparsa ed un vettore, che sia in grado di calcolare:

$$\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$$

La matrice è memorizzata nei formati:

- CSR
- HLL

Il nucleo è stato parallelizzato tramite OpenMP e CUDA ed è stato collaudato confrontando i risultati paralleli con quelli ottenuti da un’implementazione seriale. Il documento illustra in modo dettagliato le fasi di sviluppo del progetto, le scelte metodologiche adottate e l’analisi dei risultati ottenuti.

I. INTRODUZIONE

Una matrice si definisce sparsa quando la maggior parte dei suoi elementi è pari a zero. Il prodotto tra una matrice di questo tipo e un vettore è un’operazione fondamentale in numerosi ambiti scientifici. Tuttavia, nonostante la sua apparente semplicità, le caratteristiche delle matrici sparse possono compromettere l’efficienza di questo calcolo, portando ad alcuni problemi prestazionali. Tra questi si evidenziano un basso rapporto tra operazioni in virgola mobile e accessi in memoria, un’elevata dipendenza dalla larghezza di banda della memoria e l’utilizzo di indirizzamenti indiretti, che limitano la località spaziale e temporale dei dati. In questi casi, è opportuno scegliere un formato di memorizzazione dati che consenta di ridurre significativamente lo spazio occupato in memoria e di ottimizzare le prestazioni computazionali. In questo scenario la parallelizzazione dell’algoritmo rappresenta un’ulteriore strategia efficace. Suddividendo il carico

computazionale tra più unità di calcolo, è possibile mitigare i problemi di latenza della memoria e sfruttare meglio le architetture hardware moderne, sia in ambito multicore che su GPU.

II. FORMATI DI MEMORIZZAZIONE

Il progetto prevede tre rappresentazioni in memoria delle matrici sparse:

- COO
- CSR
- HLL

Le matrici vengono inizialmente lette e memorizzate nel formato COO, per poi essere convertite nei formati CSR e HLL. Il prodotto seriale tra matrice sparsa e vettore è stato implementato utilizzando il formato CSR, mentre per il calcolo parallelo è stata realizzata sia la versione in CSR che in HLL. Di seguito la descrizione dei formati indicati.

A. COO

Il formato di memorizzazione COO (Coordinate Format) rappresenta una matrice sparsa attraverso una tripla di vettori che registrano esplicitamente le coordinate degli elementi non nulli (indice di riga e di colonna) e gli elementi stessi, come mostrato in *Figura 1*. Tuttavia, questa rappresentazione comporta un indirizzamento indiretto durante l’accesso agli elementi, il quale non consente di sfruttare il principio di località dei dati. Di conseguenza, gli accessi al vettore dei valori risultano non sequenziali in memoria, con potenziali penalizzazioni in termini di prestazioni. Per questo motivo, il formato COO non è generalmente indicato per operazioni intensive come il prodotto matrice-vettore.

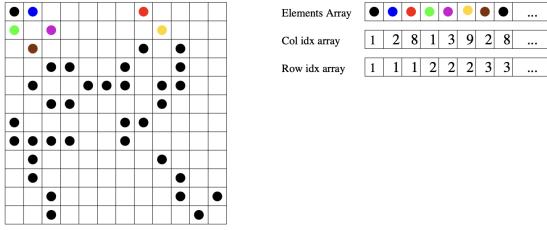


Figura 1. Rappresentazione di una matrice nel formato COO

B. CSR

Il formato di memorizzazione CSR (Compressed Sparse Row) rappresenta una matrice sparsa in modo più efficiente rispetto al formato COO, in quanto raggruppa gli elementi appartenenti alla stessa riga, evitando così di ripeterne gli indici. In CSR, la matrice è memorizzata attraverso tre vettori: uno che contiene tutti i valori non nulli (AS), uno con i corrispondenti indici di colonna (JA), e un terzo vettore ausiliario che specifica gli offset di inizio e fine di ciascuna riga all'interno dei primi due vettori (IRP). La rappresentazione è mostrata in *Figura 2*. Questa struttura consente un accesso diretto e localizzato agli elementi di ciascuna riga, migliorando la località dei dati e riducendo l'overhead computazionale. Il formato CSR è quindi particolarmente adatto a operazioni come il prodotto matrice-vettore, soprattutto nel caso di matrici di grandi dimensioni e con distribuzione irregolare degli zeri.

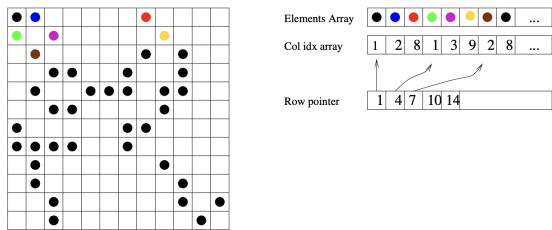


Figura 2. Rappresentazione di una matrice nel formato CSR

C. HLL

Il formato di memorizzazione HLL (Hybrid Linked List) partiziona la matrice in blocchi composti

da un numero fisso di righe, definito dal parametro *HackSize*; nel nostro caso è configurato in modo statico a 32 righe. Ciascun blocco viene successivamente memorizzato utilizzando il formato ELLPACK, mostrato in *Figura 3*, il quale prevede la costruzione di due matrici dense bidimensionali: una contenente gli indici di colonna degli elementi non nulli (JA) e l'altra i corrispondenti valori (AS). Per garantire una struttura regolare, le righe che presentano un numero di elementi non nulli inferiore al massimo osservato nel blocco vengono completate con padding (elementi pari a zero). Il formato ELLPACK consente di ottenere accessi coalizzati alla memoria globale nelle architetture GPU, condizione necessaria per sfruttare appieno la banda disponibile e ridurre la latenza. Tuttavia, ciò è possibile solo se si garantisce che gli elementi appartenenti alla stessa colonna della rappresentazione ELLPACK siano adiacenti in memoria, il che, in linguaggi con memorizzazione per riga come C o C++, implica una disposizione trasposta dei dati.

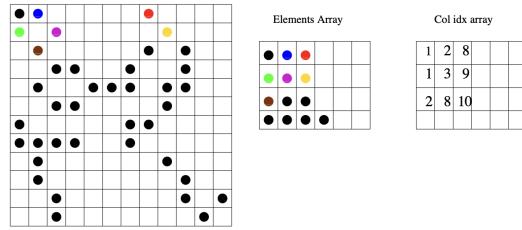


Figura 3. Rappresentazione di una matrice nel formato ELLPACK

III. PREPROCESSAMENTO

Per il collaudo del nucleo sono state usate delle matrici disponibili dalla *Suite Sparse Matrix Collection*, ciascuna caratterizzata da un numero di righe M, numero di colonne N e numero di non zeri NZ. Le matrici sono state lette nel formato Matrix Market sfruttando le funzioni disponibili nella libreria *ANSI C Matrix Market I/O*, nei file mmio.c e mmio.h.

Il preprocessamento della matrice è stato effettuato tramite la funzione *read_matrix* che, dopo aver letto la matrice dal file .mtx, si occupa di verificare tramite la lettura del banner se si tratta

di una matrice simmetrica o pattern. Qualora la matrice risulti simmetrica, nel file `.mtx` è memorizzato solo un triangolo, quindi viene predisposta la ricostruzione esplicita del triangolo mancante, con conseguente ricalcolo del numero di non zeri. Per quanto riguarda le matrici pattern, nel file `.mtx` sono memorizzate solo le coordinate dei non zeri, per cui risulta necessario esplicitare i valori ponendoli tutti pari a 1.0.

A seguito della lettura, la matrice viene memorizzata nel formato COO, allocando la seguente struttura:

```
typedef struct {
    int row, col;
    double value;
} COOElement;

typedef struct {
    int M, N, nz;
    COOElement *matrix;
} MatrixElement;
```

Tale rappresentazione, sebbene semplice e flessibile, non garantisce alcun ordinamento implicito degli elementi. Per ottenere una struttura dati coerente e preparare la matrice alla successiva conversione nei formati CSR e HLL, è stato effettuato un ordinamento lessicografico degli elementi COO, prima per riga e successivamente per colonna, implementato mediante una funzione `qsort`. Entrambe le funzioni di conversione, nel formato CSR e HLL, assumono che la matrice iniziale sia ordinata per riga al fine di popolare correttamente le strutture dati.

La matrice nel formato CSR viene memorizzata nella struttura:

```
typedef struct {
    int *IRP;
    int *JA;
    double *AS;
} CSRMatrix;
```

La matrice nel formato HLL viene memorizzata nella struttura:

```
typedef struct {
    int block_rows;
```

```
int N;
int maxnz;
int *JA;
double *AS;
} EllpackBlock;

typedef struct {
    int hackSize;
    int numBlocks;
    EllpackBlock *blocks;
} HLLMatrix;
```

Nel formato ELLPACK originale le strutture JA e AS sono matrici bidimensionali. Tuttavia, in questa implementazione sono state rappresentate come vettori monodimensionali trasposti, ovvero memorizzati per colonne anziché per righe. Questa scelta consente un accesso più efficiente alla memoria, in particolare accessi coalescenti su architetture parallele come le GPU.

Il vettore `x` impiegato nel prodotto è stato inizializzato con valori pseudo-casuali generati mediante la funzione `rand`, con distribuzione uniforme nell'intervallo [0.1, 2]. Per garantire la riproducibilità degli esperimenti, il generatore di numeri pseudo-casuali è stato inizializzato con un seme deterministico, impostato a 1234 tramite la funzione `srand`.

IV. PRODOTTO SERIALE

Il prodotto tra matrice sparsa e vettore è stato inizialmente implementato in modo seriale, assumendo che la matrice fosse memorizzata nel formato CSR. Lo pseudocodice è descritto nell'*Algoritmo 1*.

V. OPENMP

OpenMP (Open Multi-Processing) è un'API multi-threading progettata per la realizzazione di applicazioni parallele su architetture a memoria condivisa. Nel presente progetto, è stata utilizzata per parallelizzare il prodotto tra una matrice sparsa e un vettore, suddividendo le iterazioni del ciclo esterno tra i thread disponibili. OpenMP consente di specificare il numero massimo di thread da impiegare nell'esecuzione parallela. A tal fine, è stata condotta un'analisi delle prestazioni facendo variare il numero di thread da 1 fino al numero

Algorithm 1 Prodotto seriale matrice sparsa-vettore con formato CSR

Input: vettore x , matrice sparsa A nel formato CSR (IRP, JA, AS), numero di righe M
Output: vettore risultato y

```
1: for  $i = 0$  to  $M - 1$  do
2:    $sum \leftarrow 0$ 
3:   for  $j = IRP(i)$  to  $IRP(i + 1) - 1$  do
4:      $sum \leftarrow sum + AS \cdot x(JA(j))$ 
5:   end for
6:    $y(i) \leftarrow sum$ 
7: end for
```

massimo di core disponibili sulla piattaforma di calcolo adottata, nel nostro caso pari a 40. Per ciascun formato di memorizzazione della matrice sparsa, sono state sviluppate due varianti dell'algoritmo di prodotto matrice-vettore. La prima utilizza la politica di scheduling guided fornita da OpenMP, che assegna inizialmente blocchi di iterazioni di dimensione maggiore ai thread, riducendo progressivamente la dimensione dei blocchi nelle assegnazioni successive. Questo approccio permette un bilanciamento dinamico del carico di lavoro: i thread che terminano precocemente possono ricevere ulteriori porzioni di lavoro, contribuendo a una migliore distribuzione complessiva del carico. La seconda implementazione, invece, adotta una suddivisione statica del lavoro tra i thread. In questo caso, la ripartizione delle righe della matrice viene calcolata a priori, assegnando a ciascun thread una porzione fissa del lavoro, senza delegare la gestione del bilanciamento alla piattaforma OpenMP.

A. CSR

Nel caso del formato CSR, l'unità di lavoro distribuita tra i thread corrisponde alle righe della matrice. Lo pseudocodice della versione con scheduling guided nel formato CSR del prodotto è mostrato nell'*Algoritmo 2*.

Nella seconda implementazione del prodotto, attraverso la funzione `compute_row_bounds` si suddividono le righe della matrice CSR in intervalli bilanciati tra i thread, in base alla distribuzione

Algorithm 2 Prodotto parallelo matrice sparsa-vettore con formato CSR in OpenMP e scheduling guided

Input: vettore x , matrice sparsa A nel formato CSR (IRP, JA, AS), numero di righe M
Output: vettore risultato y

```
1: parallel for  $i = 0$  to  $M - 1$  con scheduling guided
2:    $sum \leftarrow 0$ 
3:   for  $j = IRP(i)$  to  $IRP(i + 1) - 1$ 
4:      $sum \leftarrow sum + AS(j) \cdot x(JA(j))$ 
5:   end for
6:    $y(i) \leftarrow sum$ 
7: end for
```

cumulativa degli elementi non nulli:

$$nz_per_thread = \left\lceil \frac{NZ}{num_threads} \right\rceil$$

Il numero totale di non zeri è suddiviso equamente tra i thread. Per ogni thread, si determina l'indice di riga corrispondente al target cumulativo degli elementi non nulli mediante una ricerca binaria nella struttura IRP. Negli *Algoritmi 3* e *4* è possibile osservare, rispettivamente, lo pseudocodice di `compute_row_bounds` e del prodotto parallelo.

Algorithm 3 Calcolo `row_bounds` per la suddivisione bilanciata delle righe CSR tra thread

Input: matrice sparsa A nel formato CSR (IRP), numero di righe M , numero di thread $num_threads$
Output: array `row_bounds` con i limiti delle righe per ogni thread

```
1:  $NZ \leftarrow IRP(M)$ 
2:  $nz\_per\_thread \leftarrow \left\lceil \frac{NZ}{num\_threads} \right\rceil$ 
3:  $row\_bounds(0) \leftarrow 0$ 
4: for  $t = 1$  to  $num\_threads - 1$  do
5:    $target \leftarrow t \cdot nz\_per\_thread$ 
6:    $row\_bounds(t) \leftarrow binary\_search(IRP, M + 1, target)$ 
7: end for
8:  $row\_bounds(num\_threads) \leftarrow M$ 
```

Algorithm 4 Prodotto parallelo matrice sparsa-vettore con formato CSR in OpenMP e suddivisione tramite `row_bounds`

Input: vettore x , matrice sparsa A nel formato CSR (IRP, JA, AS), vettore `row_bounds`, numero di righe M

Output: vettore risultato y

- 1: $num_threads \leftarrow$ numero di thread OpenMP
- 2: **parallel for** $tid = 0$ to $num_threads - 1$
- 3: **for** $i = row_bounds(tid)$ to $row_bounds(tid + 1) - 1$
- 4: $sum \leftarrow 0$
- 5: **for** $j = IRP(i)$ to $IRP(i + 1) - 1$
- 6: $sum \leftarrow sum + AS(j) \cdot x(JA(j))$
- 7: **end for**
- 8: $y(i) \leftarrow sum$
- 9: **end for**
- 10: **end for**

B. HLL

Nel caso del formato HLL, l'unità di lavoro distribuita tra i thread corrisponde ai blocchi ELLPACK della matrice. Lo pseudocodice della versione con scheduling guided nel formato HLL del prodotto è mostrato nell'*Algoritmo 5*.

La funzione `compute_block_bounds` gestisce la suddivisione dei blocchi della matrice tra i thread disponibili, assegnandoli in modo equo:

$$\text{block_per_thread} = \left\lfloor \frac{\text{num_blocks}}{\text{num_threads}} \right\rfloor$$

Le rimanenze vengono distribuite tra i primi thread, garantendo un bilanciamento ottimale del carico. Nel prodotto parallelo ogni thread elabora un sottoinsieme disgiunto di blocchi assegnati tramite gli estremi indicizzati nel vettore `block_bounds`. Negli *Algoritmi 6* e *7* è possibile osservare, rispettivamente, lo pseudocodice di `compute_block_bounds` e del prodotto parallelo.

VI. CUDA

CUDA (Compute Unified Device Architecture) è un'architettura di elaborazione parallela sviluppata da NVIDIA, che consente di sfruttare la potenza

Algorithm 5 Prodotto matrice sparsa-vettore con formato HLL in OpenMP con scheduling guided

Input: vettore x , matrice sparsa A nel formato HLL ($hack_size, num_blocks, blocks$)

Output: vettore risultato y

- 1: **parallel for** $b = 0$ to $num_blocks - 1$ **con scheduling guided**
- 2: $rows, max_nz, JA, AS \leftarrow$ elementi del blocco corrente b
- 3: $base \leftarrow b \cdot hack_size$
- 4: **for** $i = 0$ to $rows - 1$
- 5: $sum \leftarrow 0$
- 6: $global_row \leftarrow base + i$
- 7: **for** $j = 0$ to $max_nz - 1$
- 8: $idx \leftarrow j \cdot rows + i$
- 9: $col \leftarrow JA(idx)$
- 10: **if** $col \neq -1$ **then** $sum \leftarrow sum + AS(idx) \cdot x(col)$
- 11: **end for**
- 12: $y(global_row) \leftarrow sum$
- 13: **end for**
- 14: **end for**

Algorithm 6 Calcolo `block_bounds` per la suddivisione bilanciata dei blocchi ELLPACK tra thread

Input: numero di blocchi num_blocks , numero di thread $num_threads$

Output: array `block_bounds` con i limiti dei blocchi per ogni thread

- 1: $blocks_per_thread \leftarrow \left\lfloor \frac{\text{num_blocks}}{\text{num_threads}} \right\rfloor$
- 2: $remainder \leftarrow num_blocks \bmod num_threads$
- 3: $block_bounds(0) \leftarrow 0$
- 4: **for** $t = 0$ to $num_threads - 1$ **do**
- 5: $extra \leftarrow \begin{cases} 1 & \text{se } t < remainder \\ 0 & \text{altrimenti} \end{cases}$
- 6: $block_bounds(t+1) \leftarrow block_bounds(t) + blocks_per_thread + extra$
- 7: **end for**

Algorithm 7 Prodotto matrice sparsa-vettore con formato HLL in OpenMP suddivisione tramite `block_bounds`

Input: vettore x , matrice sparsa A nel formato HLL ($hack_size, blocks$), limiti dei blocchi `block_bounds`

Output: vettore risultato y

```

1: num_threads  $\leftarrow$  numero di thread OpenMP
2: parallel for  $tid = 0$  to  $num\_threads - 1$ 
3:   for  $b = block\_bounds[tid]$  to  $block\_bounds[tid + 1] - 1$ 
4:      $rows, max\_nz, JA, AS \leftarrow$  elementi del blocco corrente  $b$ 
5:      $base \leftarrow b \cdot hack\_size$ 
6:     for  $i = 0$  to  $rows - 1$ 
7:        $sum \leftarrow 0$ 
8:        $global\_row \leftarrow base + i$ 
9:       for  $j = 0$  to  $max\_nz - 1$ 
10:       $idx \leftarrow j \cdot rows + i$ 
11:       $col \leftarrow JA(idx)$ 
12:      if  $col \neq -1$  then  $sum \leftarrow sum + AS(idx) \cdot x(col)$ 
13:      end for
14:       $y(global\_row) \leftarrow sum$ 
15:    end for
16:  end for
17: end for

```

computazionale delle GPU per operazioni ad alto grado di parallelismo. L'organizzazione del calcolo segue una gerarchia composta da thread, blocchi di thread e griglie di blocchi. L'esecuzione avviene interamente sulla GPU: i dati (come matrici e vettori) vengono inizialmente trasferiti dalla memoria dell'host a quella del device, dopodiché il kernel viene lanciato specificando la configurazione del numero di blocchi e dei thread per blocco. Ciascun thread calcola il proprio indice globale in base alla propria posizione all'interno del blocco e alla posizione del blocco nella griglia, accedendo così ai dati di competenza. Per ciascun formato di memorizzazione di matrici sparse, sono state sviluppate due varianti dell'algoritmo di prodotto matrice-vettore. La prima prevede l'assegnazione di una riga della matrice a ciascun thread. In questo

caso il numero di blocchi per l'invocazione del kernel viene calcolato come segue:

$$\text{blocks} = \left\lceil \frac{\text{total_rows}}{\text{threads_per_block}} \right\rceil$$

La seconda, invece, sfrutta il concetto di warp, l'unità minima di esecuzione della GPU, composta da 32 thread. In questo caso, un'intera riga della matrice viene gestita da un singolo warp, soluzione particolarmente efficace in presenza di righe con un elevato numero di elementi non nulli. All'interno del warp, ogni thread elabora colonne diverse della riga mediante uno stride pari alla dimensione del warp. Al termine dell'elaborazione, i contributi parziali vengono aggregati attraverso una riduzione a livello di warp di tipo interleaved addressing, implementata con l'istruzione `__shfl_down_sync` che consente di sommare i contributi dei thread in maniera efficiente, evitando il ricorso alla memoria condivisa. L'approccio interleaved addressing prevede che, ad ogni iterazione, i thread con indice i sommino il proprio valore con quello del thread $i + offset$, dove $offset$ si riduce progressivamente secondo potenze di due (16, 8, 4, 2, 1). In tal modo, i contributi vengono aggregati in maniera gerarchica, dimezzando a ogni passo il numero di thread attivi e garantendo un'efficiente riduzione in tempo logaritmico rispetto alla dimensione del warp. In questo caso il numero di blocchi per l'invocazione del kernel viene calcolato come segue:

$$\text{blocks} = \left\lceil \frac{\text{total_rows} \cdot \text{warp_size}}{\text{threads_per_block}} \right\rceil$$

A. CSR

Negli *Algoritmi 8 e 9* vengono mostrati i kernel del prodotto matrice-vettore con il formato CSR, rispettivamente nella versione un thread per riga e un warp per riga.

B. HLL

Negli *Algoritmi 10 e 11* vengono mostrati i kernel del prodotto matrice-vettore con il formato HLL, rispettivamente nella versione un thread per riga e un warp per riga.

Algorithm 8 Kernel prodotto matrice sparsa-vettore con formato CSR in CUDA - un thread per riga

Input: vettore x , matrice sparsa A nel formato CSR (IRP, JA, AS), numero di righe M
Output: vettore risultato y

- 1: $row \leftarrow blockIdx.x \cdot blockDim.x + threadIdx.x$
- 2: **if** $row < M$ **then**
- 3: $sum \leftarrow 0$
- 4: **for** $j = IRP(row)$ to $IRP(row + 1) - 1$
- 5: $sum \leftarrow sum + AS(j) \cdot x(JA(j))$
- 6: **end for**
- 7: $y(row) \leftarrow sum$

Algorithm 9 Kernel prodotto matrice sparsa-vettore con formato CSR in CUDA - un warp per riga

Input: vettore x , matrice sparsa A nel formato CSR (IRP, JA, AS), numero di righe M
Output: vettore risultato y

- 1: $tid \leftarrow blockIdx.x \cdot blockDim.x + threadIdx.x$
- 2: $row \leftarrow \left\lfloor \frac{tid}{warp_size} \right\rfloor$
- 3: $lane \leftarrow threadIdx.x \bmod warp_size$
- 4: **if** $row < M$ **then**
- 5: $sum \leftarrow 0$
- 6: **for** $j = IRP(row) + lane$ to $IRP(row + 1) - 1$ **step** $warp_size$
- 7: $sum \leftarrow sum + AS(j) \cdot x(JA(j))$
- 8: **end for**
- 9: **for** $offset = warp_size/2$ to 1 **step** $offset/2$
- 10: $sum \leftarrow sum + shfl_down_sync(sum, offset)$
- 11: **end for**
- 12: **if** $lane = 0$ **then** $y(row) \leftarrow sum$

VII. MISURAZIONE DELLE PRESTAZIONI

Tutte le misurazioni sono una media di risultati ottenuti invocando ripetutamente il nucleo di calcolo, al fine di ottenere una misura più affidabile delle prestazioni. Si è scelto di effettuare 40 ripetizioni complessive, escludendo però le prime 10 dal calcolo della media. Questa scelta è motivata dalla

Algorithm 10 Kernel prodotto matrice sparsa-vettore con formato HLL in CUDA - un thread per riga

Input: vettore x , matrice A nel formato HLL ($hack_size, d_blocks$), numero totale di righe $total_rows$
Output: vettore risultato y

- 1: $global_row \leftarrow blockIdx.x \cdot blockDim.x + threadIdx.x$
- 2: **if** $globalRow \geq total_rows$ **then return**
- 3: $b \leftarrow \left\lfloor \frac{global_row}{hack_size} \right\rfloor$
- 4: $rows, max_nz, JA, AS \leftarrow$ elementi del blocco corrente b
- 5: $local_row \leftarrow global_row \bmod hack_size$
- 6: **if** $local_row \geq rows$ **then return**
- 7: $sum \leftarrow 0$
- 8: **for** $j = 0$ to $max_nz - 1$
- 9: $idx \leftarrow j \cdot rows + local_row$
- 10: $col \leftarrow JA(idx)$
- 11: **if** $col \neq -1$ **then**
- 12: $sum \leftarrow sum + AS(idx) \cdot x(col)$
- 13: **end for**
- 14: $y(global_row) \leftarrow sum$

necessità di ridurre l'influenza dei tempi di set-up e di eventuali effetti transitori iniziali sul risultato finale, che non riflettono le prestazioni effettive del nucleo in condizioni stazionarie. Le misurazioni riportate si riferiscono esclusivamente al tempo necessario per l'esecuzione del calcolo numerico del prodotto matrice-vettore, escludendo i tempi di I/O, il preprocessamento dei dati, e il trasferimento dati da e verso la memoria della GPU (nel caso CUDA). Fanno eccezione le implementazioni in OpenMP che includono il calcolo dei bound per la suddivisione del carico: in tal caso è stata condotta un'analisi separata per includere anche i tempi di preprocessamento necessari a determinare la partizione del lavoro tra i thread. Le prestazioni del calcolo sono state valutate utilizzando le seguenti metriche:

- **Flops:** rappresentano il numero di operazioni in virgola mobile eseguite al secondo, calcolati

Algorithm 11 Kernel prodotto matrice sparsavettore con formato HLL in CUDA - un warp per riga

Input: vettore x , matrice A nel formato HLL ($hack_size, d_blocks$), numero totale di righe $total_rows$

Output: vettore risultato y

- 1: $tid \leftarrow blockIdx.x \cdot blockDim.x + threadIdx.x$
- 2: $warp_id \leftarrow \left\lfloor \frac{tid}{warp_size} \right\rfloor$
- 3: $lane \leftarrow threadIdx.x \bmod warp_size$
- 4: **if** $warp_id \geq total_rows$ **then return**
- 5: $b \leftarrow \left\lfloor \frac{warp_id}{hack_size} \right\rfloor$
- 6: $rows, max_nz, JA, AS \leftarrow$ elementi del blocco corrente b
- 7: $local_row \leftarrow warp_id \bmod hack_size$
- 8: **if** $local_row \geq rows$ **then return**
- 9: $sum \leftarrow 0$
- 10: **for** $j = lane$ to $max_nz - 1$ **step** $warp_size$
- 11: $idx \leftarrow j \cdot rows + local_row$
- 12: $col \leftarrow JA(idx)$
- 13: **if** $col \neq -1$ **then**
- 14: $sum \leftarrow sum + AS(idx) \cdot x(col)$
- 15: **end for**
- 16: **for** $offset = warp_size/2$ to 1 **step** $offset/2 = 2$
- 17: $sum \leftarrow sum + __shfl_down_sync(sum, offset)$
- 18: **end for**
- 19: **if** $lane = 0$ **then** $y(warp_id) \leftarrow sum$

come:

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

dove NZ è il numero di elementi non nulli della matrice e T è il tempo medio di esecuzione espresso in secondi. I risultati sono riportati in *GigaFlops* (10^9 FLOPS).

- **Speedup:** quantifica il miglioramento ottenuto in termini di tempo di esecuzione dell'algoritmo parallelo rispetto alla versione seriale.

Viene calcolato come:

$$\text{Speedup} = \frac{T_s}{T_p}$$

dove T_s è il tempo di esecuzione seriale e T_p quello parallelo.

- **Efficienza:** misura quanto efficacemente vengono utilizzati i thread a disposizione, ossia quanto stiamo sfruttando il parallelismo. Viene calcolata come:

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

Un'efficienza pari a 1 (o 100%) indica un uso ottimale delle risorse computazionali.

VIII. ANALISI DEI RISULTATI

Sulla base delle metriche illustrate, sono state raccolte le prestazioni delle diverse versioni del nucleo di calcolo sviluppato. L'analisi è stata condotta su un'ampia selezione di matrici di test, al fine di garantire una valutazione robusta e significativa dei risultati ottenuti. Le informazioni raccolte sono state poi elaborate e visualizzate mediante grafici, generati attraverso specifiche funzioni Python, con l'obiettivo di facilitare il confronto e l'interpretazione delle varie soluzioni. Per la visualizzazione le matrici sono state suddivise in due gruppi: nel grafico superiore sono mostrate le matrici con un numero di elementi non nulli minore di 10^6 , mentre nel grafico inferiore quelle con un numero maggiore.

È stata verificata la correttezza del calcolo parallelo, sia in OpenMP che in CUDA, confrontando il risultato con quello ottenuto dall'implementazione seriale. Per ogni elemento del vettore risultante sono state misurate sia la differenza assoluta sia quella relativa, rispetto al valore seriale di riferimento. Poiché l'aritmetica floating-point non è associativa, le diverse modalità di somma in ambiente parallelo possono introdurre leggere discrepanze. È stato quindi adottato un criterio di tolleranza: una differenza assoluta inferiore a 10^{-9} e una differenza relativa inferiore a 10^{-6} sono state considerate accettabili.

A. OpenMP

Nella versione con parallelizzazione OpenMP è stata condotta un'analisi delle prestazioni al variare del numero di thread, considerando separatamente i formati di memorizzazione CSR e HLL. Sono state valutate come metriche lo speedup, i gigaflops e l'efficienza, con l'obiettivo di individuare la configurazione ottimale, ovvero il numero di thread capace di massimizzare le prestazioni. I risultati sono illustrati nelle Figure 4, 5, 6, 7, 8 e 9.



Figura 4. AvgGFlops al variare del numero di thread con formato CSR

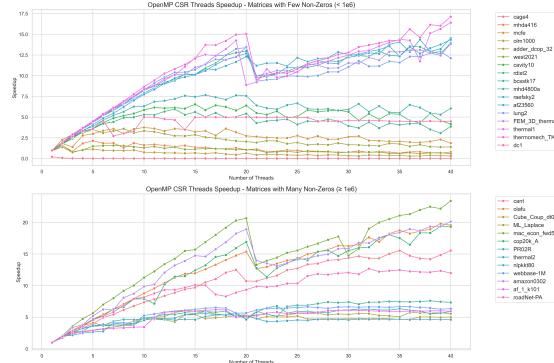


Figura 5. Speedup al variare del numero di thread con formato CSR

L'analisi dei risultati evidenzia che le migliori prestazioni, considerando una combinazione di speedup, gigaflops ed efficienza, si ottengono utilizzando circa 20 thread. Anche utilizzando il massimo

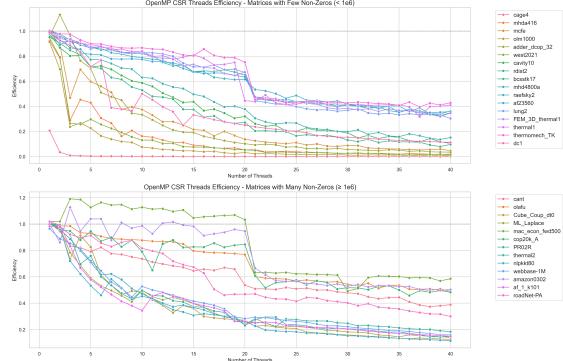


Figura 6. Efficienza al variare del numero di thread con formato CSR

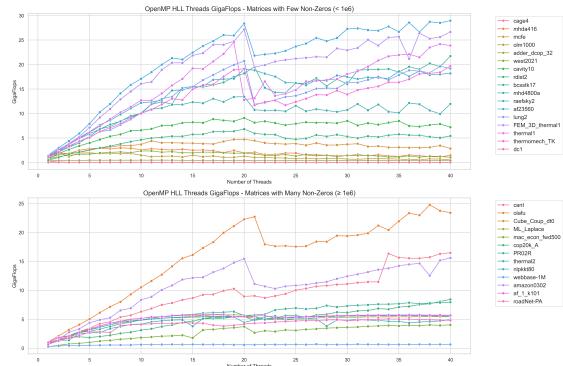


Figura 7. AvgGFlops al variare del numero di thread con formato HLL

numero di thread disponibili sulla piattaforma, cioè 40, si ottengono delle buone prestazioni in termini di gigaflops e speedup, ma l'efficienza risulta sensibilmente ridotta. Considerando infatti una visione più attenta al costo delle risorse computazionali, l'utilizzo di tutti i thread risulta meno vantaggioso, poiché implica un maggiore dispendio di risorse senza un corrispondente miglioramento proporzionale delle prestazioni. Per questo motivo, 20 thread si confermano come il miglior compromesso tra prestazioni elevate, efficienza e uso ottimale delle risorse.

Successivamente, sono state confrontate le due versioni del prodotto, quella che prevede scheduling guided e quella con suddivisione esplicita dei bound

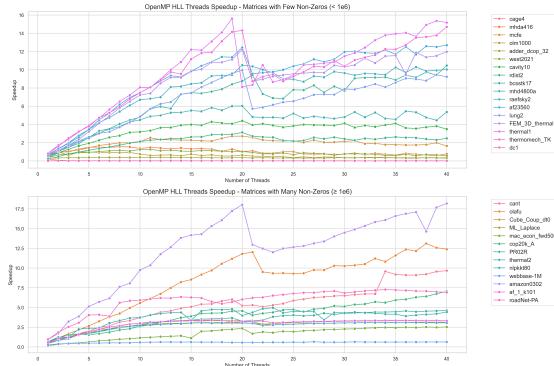


Figura 8. Speedup al variare del numero di thread con formato HLL

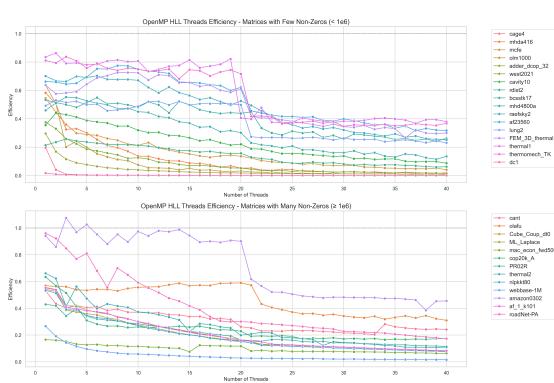


Figura 9. Efficienza al variare del numero di thread con formato HLL

di lavoro. L'obiettivo anche in questo caso è stato quello di individuare quale delle due risultasse più efficiente, utilizzando come metrica di confronto i gigaflops. I grafici relativi sono mostrati nelle Figure 10 e 11.

Dai risultati ottenuti, è emerso che, per quasi tutte le matrici analizzate e per entrambi i formati di memorizzazione, la versione con gestione esplicita dei bound offre prestazioni migliori. Questa implementazione consente di ridurre i costi di scheduling a runtime e di bilanciare in modo più efficace il carico di lavoro tra i thread, migliorando e sfruttando appieno la località dei dati in cache. Nella moltiplicazione tra matrice sparsa e vettore, infatti, le righe presentano un numero variabile di elementi

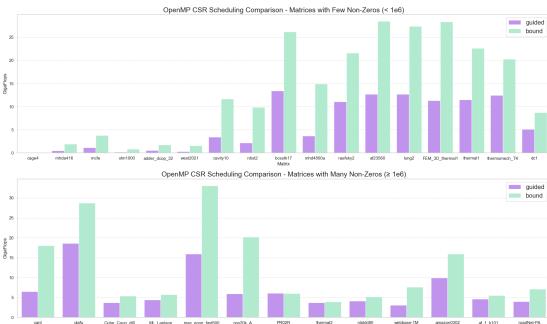


Figura 10. Confronto AvgGFlops tra scheduling guided e con bound con formato CSR

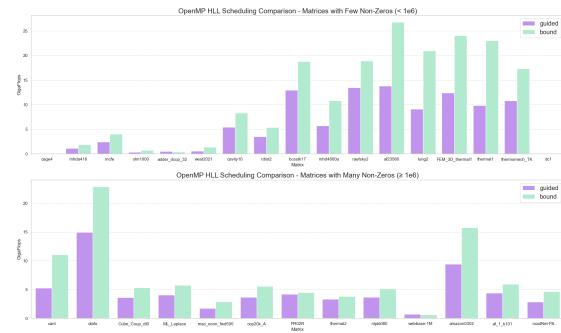


Figura 11. Confronto AvgGFlops tra scheduling guided e con bound con formato HLL

non nulli e una distribuzione irregolare dei dati, fattori che possono introdurre un significativo overhead quando il bilanciamento del carico è gestito dinamicamente.

Per garantire un confronto completo tra le due versioni, l'analisi è stata estesa includendo anche i tempi di preprocessamento necessari alla determinazione della partizione del lavoro tra i thread nella versione con bound. In questo studio, la metrica adottata è stata il tempo di esecuzione complessivo; nel caso della versione con bound, è stato quindi considerato sia il tempo impiegato per il prodotto effettivo sia quello richiesto per il calcolo della suddivisione del lavoro. I risultati di questa analisi sono riportati nelle Figure 12 e 13.

Nonostante l'aggiunta del tempo di preprocessamento, i risultati ottenuti confermano quelli dell'analisi precedente: la versione con bound continua a



Figura 12. Confronto tempo di esecuzione totale tra scheduling guided e con bound con formato CSR



Figura 13. Confronto tempo di esecuzione totale tra scheduling guided e con bound con formato HLL

offrire prestazioni migliori rispetto a quella guided, consolidando così le conclusioni già raggiunte sulla configurazione più efficiente.

È stata infine condotta un'analisi delle prestazioni adottando la configurazione ottimale individuata negli esperimenti precedenti, cioè utilizzando un massimo di 20 thread e implementando uno scheduling con bound. In particolare, sono stati confrontati i risultati ottenuti utilizzando i due formati di memorizzazione dei dati, analizzandone le prestazioni in termini di gflops e di speedup. I grafici relativi sono mostrati nelle Figure 14 e 15.

L'analisi finale evidenzia come, nella maggior parte dei casi, il formato CSR permetta di ottenere performance superiori rispetto a HLL. Questo comportamento è attribuibile alla maggiore efficienza di accesso alla memoria e alla minore complessità

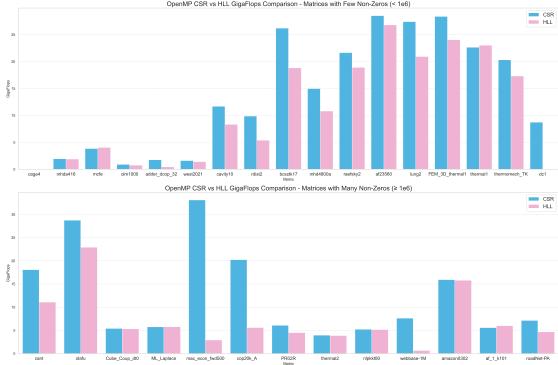


Figura 14. Confronto AvgGFlops tra formato CSR e HLL

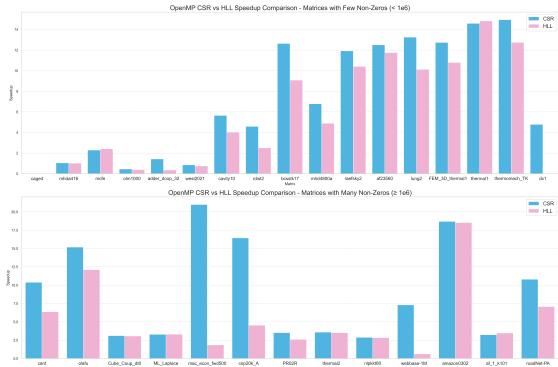


Figura 15. Confronto Speedup tra formato CSR e HLL

di gestione del formato CSR, che incide in modo determinante sulla riduzione delle latenze in ambienti paralleli. Tuttavia, è importante notare che in alcuni casi specifici, come per matrici con una distribuzione molto irregolare degli elementi non nulli, le prestazioni dei due formati possono essere comparabili.

B. CUDA

Anche nella versione parallelizzata tramite CUDA, è stato condotto uno studio per identificare la configurazione ottimale. Il primo parametro esaminato è stato il numero di thread per blocco da utilizzare nel kernel CUDA. Sono stati considerati quattro valori possibili: 128, 256, 512 e 1024 ed è stato analizzato lo speedup ottenuto in entrambe le versioni del kernel, con e senza l'uso dei warp.

L'analisi è stata effettuata su matrici rappresentate sia nel formato CSR che HLL ed è mostrata nelle Figure 16, 17, 18 e 19.

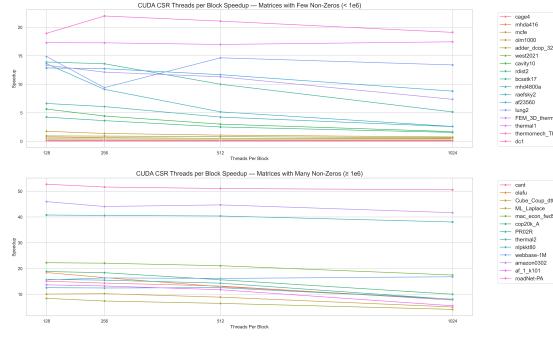


Figura 16. Speedup al variare del numero di thread per blocco nel kernel senza warp con formato CSR

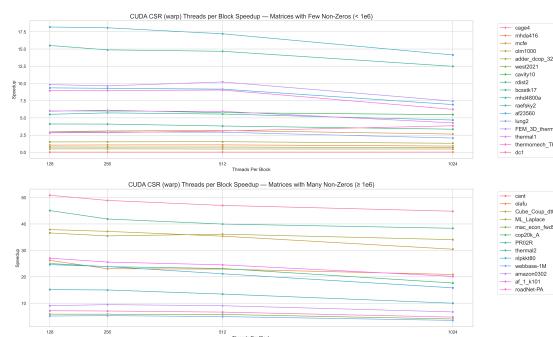


Figura 17. Speedup al variare del numero di thread per blocco nel kernel con warp con formato CSR

Dai risultati ottenuti, è emerso che il numero ottimale di thread per blocco, in media, tra tutte le diverse modalità di esecuzione, è stato 128. Questo valore ha prodotto le migliori prestazioni sia nella versione del kernel con warp che in quella senza warp con entrambi i formati di memorizzazione delle matrici. Rappresenta quindi un compromesso efficiente tra l'utilizzo delle risorse e la gestione della concorrenza, ed è stato impiegato per tutti gli studi successivi.

Sono state poi analizzate le due diverse versioni del kernel CUDA, al fine di identificare quella migliore. Il confronto è stato effettuato utilizzando

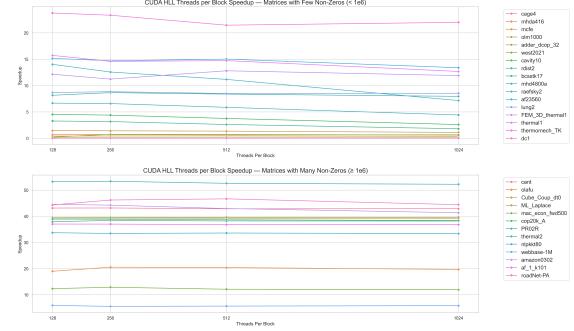


Figura 18. Speedup al variare del numero di thread per blocco nel kernel senza warp con formato HLL

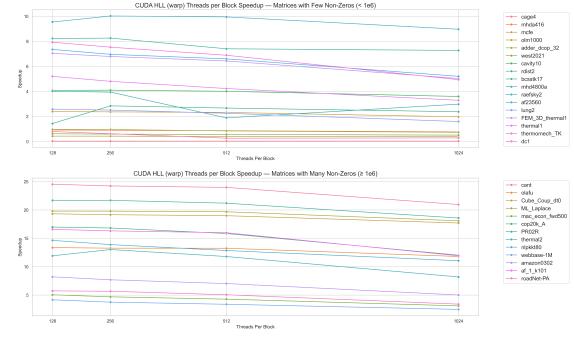


Figura 19. Speedup al variare del numero di thread per blocco nel kernel con warp con formato HLL

come metrica i gigaflops. I grafici con i risultati sono illustrati nelle Figure 20 e 21.

L'analisi è stata condotta separatamente per i formati CSR e HLL, evidenziando risultati differenti per ciascun caso. Nel formato CSR, la versione del kernel che ha offerto prestazioni superiori, per la maggior parte delle matrici, è stata quella che sfrutta l'uso dei warp, mentre nel formato HLL, l'implementazione senza warp ha prodotto risultati migliori. Questo comportamento è attribuibile alle caratteristiche strutturali dei due formati di memorizzazione. Nel CSR, la variabilità del numero di elementi non nulli per riga genera uno sbilanciamento di carico che viene efficacemente mitigato assegnando un intero warp a ciascuna riga. I thread all'interno del warp collaborano distribuendosi il lavoro e migliorando l'utilizzo della memoria e

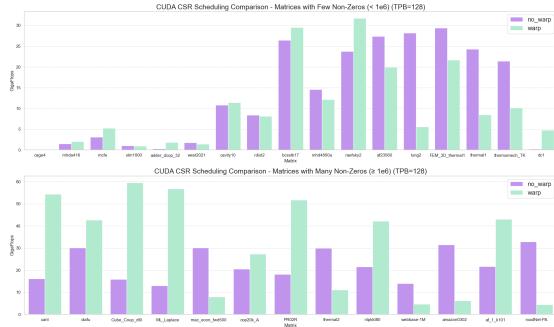


Figura 20. Confronto AvgGFlops tra kernel senza warp e con warp con formato CSR

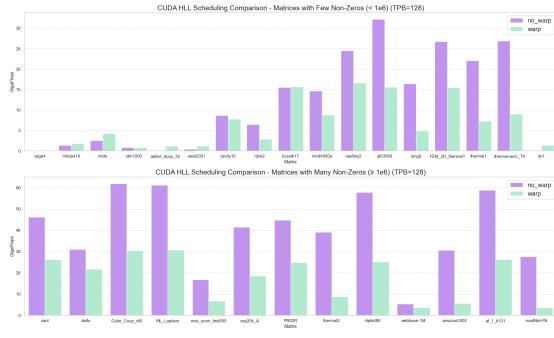


Figura 21. Confronto AvgGFlops tra kernel senza warp e con warp con formato HLL

delle unità di calcolo. Al contrario, nel formato HLL, le righe di ogni blocco sono già bilanciate artificialmente tramite padding e strutturate con un numero fisso di elementi per riga; inoltre, l’accesso coalescente ai dati consente a un singolo thread di operare con massima efficienza. In quest’ultimo caso, quindi, l’assegnazione di un warp per riga introduce un overhead di sincronizzazione, degradando le prestazioni complessive.

Infine, è stata condotta un’analisi comparativa delle prestazioni tra i due formati di memorizzazione, utilizzando come metriche i gigaflops e lo speedup. Al fine di garantire un confronto equo, per ciascun formato sono state selezionate le versioni del kernel che hanno ottenuto le migliori prestazioni sulla maggior parte delle matrici: quella che utilizza i warp per il formato CSR e quella che non li

utilizza per il formato HLL. I grafici sono mostrati nelle Figure 22 e 23.

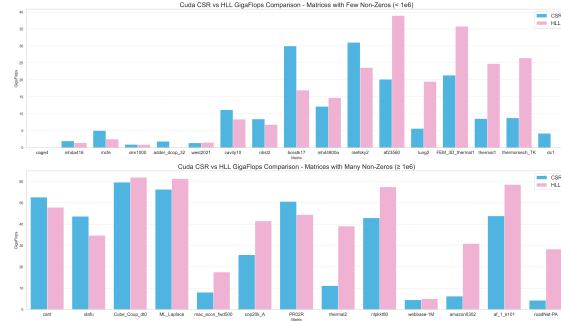


Figura 22. Confronto AvgGFlops tra formato CSR e HLL

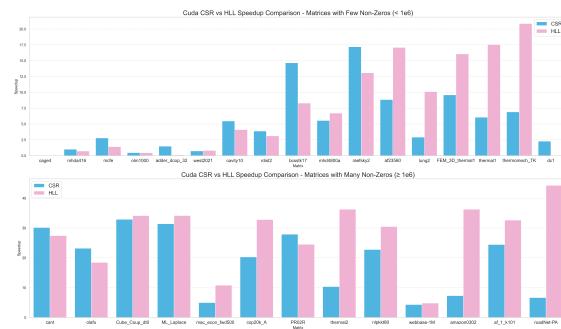


Figura 23. Confronto Speedup tra formato CSR e HLL

Dai risultati emerge che, sebbene non in modo del tutto netto, il formato CSR tende a garantire prestazioni superiori su matrici caratterizzate da un numero ridotto di elementi non nulli. Tuttavia, all’aumentare della dimensione della matrice, il formato HLL mostra prestazioni migliori sia in termini di gigaflops sia di speedup, risultando particolarmente efficace nel trattamento di matrici sparse con distribuzioni irregolari degli elementi non nulli. Questo comportamento è attribuibile all’ottimizzazione degli accessi alla memoria e alla maggiore parallelizzazione effettiva che HLL consente su architetture GPU a fronte della maggiore sensibilità del formato CSR a fenomeni di carico sbilanciato e accessi non coalescenti.

IX. SUDDIVISIONE DEL LAVORO

Il progetto è stato condotto in coppia ed il lavoro è stato uniformemente suddiviso come segue:

- Preprocessamento dei dati: Sara Malaspina, Silvia Perelli;
- Conversione della matrice nel formato CSR: Sara Malaspina;
- Conversione della matrice nel formato HLL: Silvia Perelli;
- Prodotto matrice vettore in OpenMP con formato CSR: Sara Malaspina;
- Prodotto matrice vettore in OpenMP con formato HLL: Silvia Perelli;
- Prodotto matrice vettore in CUDA con formato CSR: Silvia Perelli;
- Prodotto matrice vettore in CUDA con formato HLL: Sara Malaspina;
- Raccolta dei risultati e rappresentazione nei grafici: Sara Malaspina, Silvia Perelli;
- Stesura della relazione: Sara Malaspina, Silvia Perelli.