TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Univesità degli Studi di Roma

Tor Vergata

Parallel Sparse Matrix-Vector Product

Luca Martorelli
lucam0109@gmail.com

Alessandro Cortese
ale.cortese99@gmail.com

Abstract

Nella seguente relazione vengono descritte le fasi dello sviluppo di più nuclei di calcolo parallelo per il prodotto tra una matrice sparsa e un vettore.

I nuclei di calcolo utilizzati sono stati sviluppati per sfruttare le risorse di calcolo disponibili utilizzando la parallelizzazione OpenMP e CUDA, soluzioni sviluppate in C, che sfruttano rispettivamente le capacità di calcolo parallelo delle CPU multicore e delle GPU.

Il funzionamento e le prestazioni dei vari nuclei di calcolo vengono confrontate con una implementazione seriale minimale, sviluppata utilizzando sia il formato CSR che il formato HLL.

Contents

1 Introduzione

Il prodotto matrice sparsa-vettore (SpMV, Sparse Matrix-Vector Multiplication) è un'operazione fondamentale in numerosi ambiti scientifici e ingegneristici, tra cui la simulazione numerica, la risoluzione di sistemi lineari sparsi e l'apprendimento automatico. A differenza delle matrici dense, che memorizzano esplicitamente tutti gli elementi, le matrici sparse contengono prevalentemente zeri e sono rappresentate attraverso formati compatti per ottimizzare lo spazio di memoria e il tempo di computazione, oltre al fatto che per matrici molto grandi non si avrebbe la possibilità di rappresentarle interamente in memoria.
L'operazione SpMV si esprime come:

$$y \leftarrow Ax$$

dove A è una matrice sparsa di dimensione $M \times N$, x è un vettore denso di dimensione N, e y è il vettore risultato di dimensione M. L'efficienza del calcolo di SpMV dipende fortemente dal formato di memorizzazione della matrice, poiché esso influenza sia l'accesso ai dati sia il parallelismo computazionale sfruttabile.
I formati utilizzati per la rappresentazione della matrice in memoria, in modo da superare i problemi precedentemente indicati, sono:

- `CSR`;
- `HLL`.

Nonostante la semplicità concettuale dell'operazione di prodotto matrice-vettore, la sua implementazione efficiente presenta diverse sfide, principalmente legate alla gestione della memoria e alla natura irregolare degli accessi ai dati. In questo contesto, il parallelismo riveste un ruolo fondamentale nell'ottimizzazione delle prestazioni, consentendo di distribuire il carico computazionale su più processori, thread o unità di elaborazione. L'impiego di architetture parallele, quali CPU multicore e GPU, permette di accelerare il calcolo, garantendo al contempo la correttezza dell'operazione come requisito imprescindibile.
Le operazioni principali, quindi, sono: il recupero degli elementi non nulli della matrice e la conseguente memorizzazione nei formati `CSR` ed `HLL`, l'utilizzo dei nuclei di calcolo utilizzando OpenMP e CUDA.

1.1 Processamento dei Dati

I nuclei di calcolo sviluppati sono stati collaudati utilizzando diverse matrici sparse memorizzate in un file nel formato Matrix Market con estensione *.mtx*.
Per la gestione di questi file, è stata sviluppata una funzione dedicata all'analisi delle caratteristiche delle matrici impiegate e alla loro preparazione per l'operazione di prodotto matrice-vettore. In primo luogo, la funzione esegue la lettura del file al fine di identificare eventuali proprietà distintive della matrice, come la presenza di una struttura *pattern*, in cui i coefficienti non nulli sono implicitamente considerati pari a 1.0 e vengono rappresentati esclusivamente tramite le loro posizioni, senza esplicitarne i valori.
La funzione verifica la presenza di simmetria, che può essere superiore o inferiore. In questi casi il file memorizza soltanto una parte della matrice, omettendo la duplicazione dei valori simmetrici; più precisamente vengono registrati unicamente gli elementi appartenenti alla porzione superiore o inferiore rispetto alla diagonale principale, riducendo così lo spazio di archiviazione richiesto e ottimizzando l'elaborazione della matrice.
Una volta identificate le caratteristiche della matrice, i dati vengono estratti nel formato `(riga, colonna, valore)`. Nel caso di matrici con struttura pattern, in cui i valori non nulli non sono esplicitamente memorizzati nel file, vengono letti esclusivamente gli indici di riga e colonna, assegnando manualmente un valore predefinito pari ad 1.0.
Le informazioni raccolte vengono salvate in una struttura dati progettata per rappresentare la

matrice in una forma il più generale possibile, memorizzando tutte le proprietà rilevate durante la fase di lettura. Questa rappresentazione preliminare costituisce la base per la successiva conversione nei formati di memorizzazione `CSR` e `HLL`.

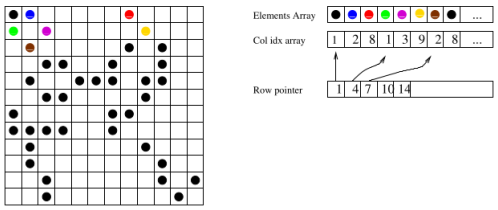
2 Rappresentazione in memoria delle Matrici Sparse

Il formato utilizzato per rappresentare le matrici è fondamentale per ridurre l'utilizzo della memoria e per poter ottimizzare le operazioni, nel caso analizzato il prodotto matrice-vettore. Come detto precedentemente, i formati utilizzati sono `CSR` ed `HLL`.

2.1 CSR

Il formato di memorizzazione `CSR` rappresenta una matrice $M \times N$ con NZ non-zeri nel seguente modo:

- **M**: numero di righe;
- **N**: numero di colonne;
- **IRP(1: M+1)**: vettore dei puntatori all'inizio di ciascuna riga;
- **JA(1:NZ)**: vettore degli indici di colonna;
- **AS(1:NZ)**: vettore dei coefficienti.



Utilizzando il seguente formato, l'operazione di prodotto di matrice-vettore può essere realizzato nel seguente modo:

Algorithm 1 SpMV(y, irp, ja, as, x)

#y: vettore risultante
#irp: vettore dei puntatori all'inizio di ciascuna riga
#ja: vettore degli indici di colonna
#as: vettore dei coefficienti
#x: vettore colonna

for $i = 1$ **to** m **do**
  $t \leftarrow 0$
  for $j = irp(i)$ **to** $irp(i+1) - 1$ **do**
    $t \leftarrow t + as(j) \times x(ja(j))$
  end for
  $y(i) \leftarrow t$
end for

2.2 HLL

Il formato `HLL` può essere definito in questo modo:

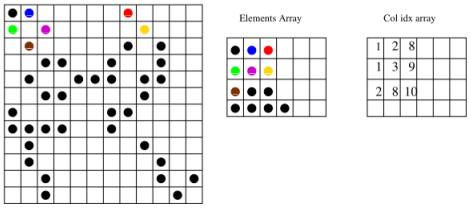
- si stabilisce un parametro **HackSize**, nel caso di studio vale 32;
- si partiziona la matrice di input in blocchi di HackSize righe;
- ciascun blocco viene memorizzato in formato **ELLPACK**, descritto nel seguito;
- i blocchi vengono memorizzati in una struttura dati.

2.2.1 ELLPACK

Una matrice $M \times N$ con NZ non-zeri e tale che $MAXNZ$ sia il massimo numero di nonzeri per riga, su tutte le righe, viene descritta come segue:

- **M**: numero di righe;
- **N**: numero di colonne;
- ***MAXNZ***: massimo di valori non-zeri per riga;
- **JA(1:M,1:MAXNZ)**: array bidimensionale di indici di colonna;
- **AS(1:M,1:MAXNZ)**: array bidimensionale di coefficienti.

Bisogna considerare che se le righe dovessero avere un numero di non-zeri inferiore a $MAXNZ$, ovvero il numero massimo di nonzeri per riga all'interno di un blocco, devono essere riempite con coefficienti appropriati, ossia zeri in AS.



Utilizzando il seguente formato, l'operazione di prodotto di matrice-vettore può essere realizzato nel seguente modo:

Algorithm 2 SpMV(y, maxnzr, ja, as, x)

#y: vettore risultante
#maxnzr: massimo di valori non-zeri per riga
#ja: vettore degli indici di colonna
#as: vettore dei coefficienti
#x: vettore colonna

for $i = 1$ **to** m **do**
  $t \leftarrow 0$
  for $j = 1$ **to** $maxnzr$ **do**
    $t \leftarrow t + as(i, j) \times x(ja(i, j))$
  end for
  $y(i) \leftarrow t$
end for

Nel caso di studio considerato il formato di memorizzazione utilizzato e implementato per la rappresentazione della matrice utilizza degli array monodimensionali e non delle matrici per tenere traccia delle informazioni riguardanti gli indici di colonna e i coefficienti della matrice.
Per il formato `HLL` è stata utilizzata la seguente struttura dati `C`:

```
typedef struct
{
    char name[256];
    int M;
    int N;
    int *offsets;
    int offsets_num;
    int *col_index;
    double *data;
    int data_num;
    int hacks_num;
    int hack_size;
    int *max_nzr;
} HLL_matrix;
```

Dove:

- `char name[256]`: rappresenta il nome della matrice;
- `M`: indica il numero di righe;
- `N`: indica il numero di colonne;
- `int *offsets`: array contenente gli offset per individuare le righe di inizio dei blocchi `HLL`, viene utilizzato per accedere più velocemente alle sezioni della matrice;
- `int offsets_num`: indica il numero di elementi in `*offsets`, ovvero indica il numero di blocchi presenti in modo da poter iterare più facilmente tra i blocchi della matrice;
- `int *col_index`: Indici delle colonne associate ai valori non nulli, viene utilizzato un approccio lineare nella memorizzazione e non uno bidimensionale per avere un minore spreco di memoria utilizzata e per mantenere un accesso sequenziale in memoria;
- `double *data`: vettore contenente i valori dei coefficienti non nulli della matrice, viene utilizzato un array monodimensionale per poter avere un accesso contiguo un memoria;
- `int data_num`: indica il numero totale di elementi in `data` e `col_index`, ovvero indica il numero di elementi non nulli della matrice,in questo modo si evita il ricalcolo del numero di elementi, con l'obiettivo di migliorare le prestazioni nei loop;
- `int hack_num`: numero di blocchi che compongono la matrice;
- `int hack_size`: dimensione di ciascun blocco `HLL`;
- `int *maz_nzr`: numero massimo di non-zeri per ogni blocco, permette un'allocazione efficiente della memoria per ogni blocco ed evita sprechi di spazio mantenendo una gestione dinamica del formato `HLL`.

L'utilizzo della precedente struttura dati viene utilizzata per una migliore gestione della memoria cercando di non aggiungere troppi coefficienti di padding, per avere un accesso ai dati più efficiente per cercare di ridurre la latenza della memoria e per mantenere un accesso sequenziale della memoria.
Si utilizza un array monodimensionale piuttosto che un array bidimensionale per avere un beneficio in termini di località spaziale della matrice, in particolar modo considerando il modo in

cui il linguaggio di programmazione `C` memorizza le matrici.
Nel caso in cui la matrice rappresentante gli indici di colonna e i coefficienti fossero memorizzate nel seguente modo `int matrix[N][N]` allora verrebbero memorizzate in memoria riga per riga, ma nel caso di studio non è possibile avere a tempo di compilazione le dimensioni delle matrici degli indici di colonna e dei coefficienti, questo perché le matrici vengono lette da file e costruite a tempo di esecuzione, quindi non è possibile utilizzare questo tipo di memorizzazione.
Nel caso in esame bisognerebbe utilizzare un formato di memorizzazione del tipo `int **matrix`, in questo caso in `C` ogni riga viene allocata separatamente causando una discontinuità nella memoria peggiorando la località a livello della cache.
Utilizzando un array monodimensionale, ovvero come nel caso della soluzione utilizzata, si mantengono tutti gli elementi contigui in memoria migliorando la località a livello della cache.
Sfruttando un array monodimensionale si ha un accesso sequenziale e prevedibile in modo da ridurre la probabilità di avere un *cache miss*, in questo modo scorrendo un array monodimensionale si avrà un pattern di accesso lineare e si avrà un beneficio anche in termini di prefetching della CPU migliorando le prestazioni.
L'utilizzo di questo formato di memorizzazione porta dei benefici anche in termini di compatibilità con CUDA, considerazioni che varranno esposte più avanti.
Utilizzando questo formato di memorizzazione, il calcolo del prodotto matrice sparsa-vettore può essere eseguito con la seguente porzione di codice:

```
1  void matvec_serial_hll(HLL_matrix *hll_matrix, double *x, double *y)
2  {
3      int rows = num_of_rows(hll_matrix, 0);
4      for (int h = 0; h < hll_matrix->hacks_num; ++h)
5      {
6          for (int r = 0; r < num_of_rows(hll_matrix, h); ++r)
7          {
8              double sum = 0.0;
9              for (int j = 0; j < hll_matrix->max_nzr[h]; ++j)
10             {
11                 int k = hll_matrix->offsets[h]+ r * hll_matrix->max_nzr[h] + j;
12                 sum += hll_matrix->data[k] * x[hll_matrix->col_index[k]];
13             }
14             y[rows * h + r] = sum;
15         }
16     }
17 }
```

3 Metriche Prestazionali

Vengono utilizzate le seguenti metriche per i risultati sperimentali:

- **Flops**: viene misurata tramite la seguente formula:

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

  Dove:

  - NZ: rappresenta il numero di non-zeri della matrice;
  - T: tempo medio di esecuzione dell'implementazione in secondi.

- **SpeedUp**: consente di confrontare l'efficienza dell'esecuzione parallela rispetto a quella seriale. Viene utilizzata la seguente formula:

$$S(W,p) = \frac{T_s(W)}{T_p(W,p)}$$

Dove:

- W: rappresenta la dimensione del problema;
- p: indica il numero di processori utilizzati;
- $T_s(W)$: indica il tempo utilizzato per l'esecuzione seriale;
- $T_p(W, p)$: indica il tempo di esecuzione parallela utilizzando p processori.

- **Efficienza**: viene misurato quanto bene vengono sfruttati le risorse di calcolo durante l'esecuzione di un algoritmo parallelo, rispetto a un'ideale esecuzione in cui ogni processore contribuisce in modo perfettamente lineare.

4 Matrici di Collaudo & Vettore X

Le matrice di collaudo utilizzate per il caso considerato sono:
cage4, mhda416, mcfe, olm1000, adder dcop 32, west2021, cavity10, rdist2, cant, olafu, Cube_Coup_dt0, ML_Laplace, bcsstk17, mac_econ_fwd500, mhd4800a, cop20k_A, raefsky2, af23560, lung2, PR02R, FEM_3D thermal1, thermal1, thermal2, thermomech_TK, nlpkkt80 webbase-1M, dc1, amazon0302, af_1_k101, roadNet-PA.
Il vettore utilizzato per il prodotto viene generato pseudo-randomicamente con una dimensione opportuna in modo tale da poter effettuare il prodotto matrice-vettore. Il vettore viene generato utilizzando la seguente porzione di codice

```
for (int i = 0; i < size; i++)
    x[i] = (rand() % 5) + 1;
```

La generazione dei coefficienti del vettore sono stati generati in questo modo per evitare problemi di rappresentazione numerica: lavorare con numeri in virgola mobile in `C` potrebbe introdurre errori di arrotondamento dovuti alla rappresentazione binaria e utilizzando valori piccoli, si riduce il rischio di tali errori nelle operazioni numeriche.
Utilizzando valori troppo piccoli, cioè se fossero molto vicini a zero, con il prodotto matrice-vettore si potrebbero amplificare eventuali errori numerici, soprattutto in presenza di coefficienti molto grandi o molto piccoli nella matrice sparsa.
Mantenere i valori in un intervallo moderato aiuta a migliorare la stabilità numerica delle operazioni.

5 Misurazione Dell'Errore

Per valutare la precisione dell'esecuzione parallele rispetto alla versione sequenziale si utilizza una funzione che misura l'errore del risultato appena ottenuto con un riferimento noto.
L'errore viene calcolato utilizzando la seguente formula:

$$s = \frac{1}{n} \sum_{i=0}^{n-1} \delta_i, \quad \text{con} \quad \delta_i = \begin{cases} |z_i - y_i|, & \text{se } |z_i - y_i| \geq \varepsilon \\ 0, & \text{altrimenti} \end{cases}$$

Dove:

- z è il vettore ottenuto dal calcolo parallelo del prodotto matrice-vettore;
- y è il vettore di riferimento;
- n è la dimensione del vettore;
- ϵ è una soglia di tolleranza al di sotto della quale le differenze vengono ignorate.

L'algoritmo percorre tutti gli elementi dei vettori z e y, calcolando la differenza assoluta tra gli elementi corrispondenti. Se la differenza supera la soglia ϵ, allora viene sommata a un accumulatore s. Alla fine, s viene normalizzato dividendo per n, ottenendo così una stima media dell'errore.
Se il valore di s è maggiore di zero, viene stampato il valore dell'errore e la funzione restituisce 0, indicando che l'errore è significativo. Se invece s fosse pari a 0 allora la funzione restituisce 1, indicando che il risultato calcolato con la versione parallela è numericamente equivalente al riferimento entro la tolleranza specificata.
Nel caso in esame viene utilizzato come risultato di riferimento il vettore risultante ottenuto con il calcolo seriale utilizzando il formato CSR.

6 Raccolta dei Dati

Tutti i dati ottenuti sono stati raccolti utilizzando una CPU Intel Xeon Silver 4210, per quanto riguarda l'esecuzione seriale con entrambi i formati e l'esecuzione parallela con OpenMP, e una scheda video NVIDIA RTX5000 per quanto riguarda le prestazioni ottenute utilizzando CUDA. Ogni nucleo di calcolo viene eseguito più volte in modo tale da avere una media dei tempi di esecuzione.

7 OpenMP

7.1 Introduzione

OpenMP, Open Multiprocessing, è un API multipiattaforma per la creazione di applicazioni parallele su sistemi a memoria condivisa supportata da vari linguaggi di programmazione e su varie architetture di calcolatori e sistemi operativi. È composto da un insieme di direttive di compilazione, routine di librerie e variabili d'ambiente che ne definiscono il funzionamento a run-time, rendendolo un modello portabile e scalabile che fornisce al programmatore un'interfaccia semplice e flessibile per lo sviluppo di applicazioni parallele.
Con OpenMP si ha un modello Fork-Join per coordinare i threads, in particolare si considera un thread master che esegue la sezione sequenziale del programma finché non arriva nella zona dal programma in cui è richiesta una fase concorrente dove crea un certo numero di thread worker che si occupano di eseguire la funzionalità richiesta. I thread worker vengono eseguiti in modo concorrente e il run-tyme system alloca i thread sui processori disponibili e comunicano tramite una memoria condivisa.
La porzione di codice che si intende eseguire in parallelo viene marcata attraverso delle apposite direttive che causano la conseguente creazione dei thread worker prima dell'esecuzione effettiva della porzione di codice parallela.
Un aspetto fondamentale utilizzato nel caso in esame è stato l'analisi delle prestazioni del nucleo di calcolo in funzione del numero di threads utilizzati, in particolare sono stati condotti dei test variando il numero di threads da 2 a 40, ovvero fino a raggiungere il numero massimo di threads supportati dal processore utilizzato per la raccolta dei dati. Il numero di thread effettivmante utilizzato viene impostato con una apposita API OpenMP. In questo modo è stato possibile ottenere una visione dettagliata dell'andamento delle prestazioni.
Un aspetto fondamentale da considerare con le esecuzione parallela è la distribuzione bilanciata del carico di lavoro tra i vari threads utilizzati, se questa operazione fosse eseguita erroneamente si potrebbero compromettere in modo negativo le prestazioni. L'obiettivo è quello di massimizzare l'utilizzo delle risorse computazionali a disposizione.
Per poter bilanciare il carico tra i vari threads si possono utilizzare diverse strategia, tra le quali utilizzare le direttive primitive di OpenMP oppure suddividere manualmente il carico di lavoro tra i threads.

Considerando le soluzioni proposte da OpenMP, soluzione attuata nella soluzione in esame, è quella di utilizzare la direttiva `#pragma omp parallel for schedule (SCHEDULE_TYPE)` che suddivide le iterazioni di un loop tra i thread disponibili.
Il valore `SCHEDULE_TYPE` indica come le iterazioni devono essere assegnate:

- `static`: vengono distribuite le iterazioni in blocchi di dimensione fissa assegnati staticamente ai thread;
- `dynamic`: le iterazioni vengono assegnate ai thread in modo dinamico, richiedendo nuovi blocchi di iterazioni man mano che i thread terminano quelli precedenti;
- `guided`: la strategia di bilanciamento è simile a `dynamic`, ma la dimensione dei blocchi diminuisce esponenzialmente, iniziando con blocchi più grandi per poi ridurla nel tempo;
- `auto`: la scelta della strategia di bilanciamento del carico più appropriata viene lasciata al compilatore;
- `runtime`: la strategia di scheduling viene decisa a tempo di esecuzione sulla base del valore della variabile d'ambiente `OMP_SCHEDULE`, in questo modo si ottiene una maggiore flessibilità dato che non viene modificato il codice sorgente ma viene modificata la variabile d'ambiente.

Altre direttive che possono essere utilizzate per bilanciare il carico di lavoro dei thread sono:

- `#pragma omp sections`: permette di suddividere il codice in sezioni indipendenti che possono essere eseguite in parallelo da diversi thread, dove ogni sezione è definita con la direttiva;
- `#pragma omp task`: consente la creazione di task indipendenti che il runtime di OpenMP assegna dinamicamente ai thread disponibili, in questo modo si facilita l'operazione di bilanciamento del carico in applicazioni con attività non prevedibili o irregolari.

La scelta della strategia di bilanciamento del carico e delle direttive appropriate dipende dalla natura del problema e dalla distribuzione del carico di lavoro. Per loop con iterazioni di durata uniforme, lo scheduling statico può essere efficiente, mentre per loop con iterazioni di durata variabile lo scheduling dinamico o guidato può offrire un migliore bilanciamento del carico.
L'altra strategia che è possibile utilizzare è quella di suddividere manualmente il carico di lavoro senza utilizzare le soluzioni proposte da OpenMP.
In questo modo si suddividono le righe equamente riducendo il rischio che alcuni thread abbiano molto più lavoro di altri, inoltre si può ottimizzare ulteriormente la suddivisione in base a caratteristiche specifiche del problema.
Questo modo rappresenta una valida alternativa e potrebbe essere più efficiente dell'utilizzo delle direttive fornite da OpenMP in scenari dove si hanno distribuzioni del carico sbilanciate.

7.2 Esecuzione parallela con il formato CSR

L'operazione di prodotto matrice-vettore in parallelo, utilizzando il formato `CSR`, è stata realizzata seguendo un processo suddiviso in più fasi. In particolare, per sfruttare al meglio il parallelismo, è stato adottato un approccio mirato al bilanciamento del carico di lavoro tra i thread impiegati. A tal fine, la funzione `matrix_partition` si occupa della suddivisione delle righe della matrice `CSR` tra i thread. Tuttavia, invece di assegnare le righe in modo uniforme, la suddivisione si basa sulla distribuzione degli elementi non nulli, garantendo un migliore bilanciamento del carico computazionale. Per ottenere questa ripartizione equilibrata, vengono effettuati i seguenti passaggi:

- Viene calcolato il numero di elementi non nulli per ciascuna riga (`row_non_zero_count`), sommando contestualmente anche il totale globale di elementi non nulli nella matrice (`total_non_zero_elements`).
- Si determina il carico di lavoro medio per thread `target_workload_per_thread`, dividendo il numero totale di elementi non nulli per `num_threads`.
- Le righe vengono assegnate ai thread in base al numero di elementi non nulli, cercando di mantenere il carico di ciascun thread il più vicino possibile al valore target.
- Poiché la partizione ottimale può differire da quella iniziale, il numero effettivo di thread (`actual_num_threads`) viene aggiornato per riflettere la suddivisione reale della matrice.
- L'array degli indici di partenza delle righe per thread (`temp_start_row_indices`) viene riallocato per adattarsi alla dimensione finale effettivamente utilizzata, e le strutture di supporto temporanee vengono deallocate per liberare la memoria.

La distribuzione uniforme del carico è stata dunque eseguita in tal modo:

Algorithm 3 matrix_partition(csr_matrix, num_threads, actual_num_threads)

#*csr_matrix*: matrice CSR
#*num_threads*: Numero di thread sul quale fare la partiziona
#*actual_num_threads*: Numero di threads che svolgeranno il prodotto

Inizializzazione & Allocazione Memoria:
row_non_zero_count ← Array di dimensione M per il conteggio degli elementi non nulli per riga
total_non_zero_elements ← 0
temp_start_row_indices ← Array di dimensione *num_threads* + 1, che indica da quale riga inizia il lavoro di ogni thread all'interno della matrice CSR

for $i = 1$ **to** m **do**
  $row\_non\_zero\_count[row] \leftarrow csr\_matrix.IRP[row + 1] - csr\_matrix.IRP[row]$
  $total\_non\_zero\_elements \leftarrow total\_non\_zero\_elements + row\_non\_zero\_count[row]$
end for

$target\_workload\_per\_thread \leftarrow total\_non\_zero\_elements / num\_threads$
$current\_thread\_workload \leftarrow 0$
$current\_thread\_id \leftarrow 0$
$temp\_start\_row\_indices[0] \leftarrow 0$

for $row = 0$ **to** $csr\_matrix.M - 1$ **do**
  $current\_thread\_workload \leftarrow current\_thread\_workload + row\_non\_zero\_count[row]$
  if $current\_thread\_workload \geq target\_workload\_per\_thread$ && $current\_thread\_id < num\_threads - 1$ **then**
    $current\_thread\_id \leftarrow current\_thread\_id + 1$
    $temp\_start\_row\_indices[current\_thread\_id + 1] \leftarrow row + 1$
    $current\_thread\_workload \leftarrow 0$
  end if
end for

$temp\_start\_row\_indices[current\_thread\_id + 1] \leftarrow csr\_matrix -> M$
$used\_threads \leftarrow current\_thread\_id + 1$
$new\_size \leftarrow used\_threads + 1$
$temp\_start\_row\_indices \leftarrow$ *reallocazione di temp_start_row_indices, di grandezza new_size*
$*actual\_num\_threads \leftarrow used\_threads$

Deallocazione Memoria: *row_non_zero_count*
return *temp_start_row_indices*

Dopo aver partizionato il carico di lavoro, possiamo eseguire il vero prodotto matrice-vettore utilizzando la funzione `product`, dentro la quale andremo a prendere in considerazione i realtivi tempi di esecuzione del prodotto. Tale tempo di esecuzione viene misurato con la funzione `omp_get_wtime()`. Il calcolo del prodotto matrice-vettore viene eseguito dopo aver introdotto il parallellismo tramite la direttiva `pragma omp parallel`, che consente di suddividere l'elaborazione tra più thread, permettendo l'esecuzione simultanea di più operazioni. Il numero di thread è specificato dal parametro `num_threads`.
Di seguito l'algortitmo del prodotto:

Algorithm 4 product(csr_matrix, x, y, num_threads, execution_time, first_row)

#*csr_matrix*: Matrice CSR
#x: Vettore con il quale si effettua il prodotto
#y: Vettore risultato del prodotto
#*num_threads*: Numero di thread
#*execution_time*: Tempo di esecuzione del prodotto
#$first\_row$: Vettore di assegnazione delle righe ai thread

Inizializzazione $start\_time \leftarrow Tempo\ iniziale$
Creazione di un insieme di num_threads thread per il parallelismo

$thread\_id \leftarrow ID\ del\ thread$

for $row = 1$ **to** $first\_row[thread\_id + 1] - 1$ **do**
  $sum \leftarrow 0$
  for $idx = csr\_matrix->IRP[row]$ **to** $csr\_matrix->IRP[row + 1]$ **do**
    $sum \leftarrow sum + csr\_matrix->AS[idx] * input\_vector[csr\_matrix->JA[idx]]$
  end for
  $output\_vector[row] \leftarrow sum$
end for

$end\_time \leftarrow Tempo\ finale$
$*execution\_time \leftarrow end\_time\ -\ start\_time$

7.3 Esecuzione parallela utilizzando il formato HLL

Per quanto riguarda il calcolo del prodotto tra matrice sparsa-vettore utilizzando il formato di memorizzazione `HLL` utilizzando OpenMP non è stata seguita la stessa strategia utilizzata per il formato `CSR`, in particolare il carico assegnato ai vari thread non è stato bilanciato manualmente ma è stata utilizzata la direttiva fornita da OpenMP stesso, in particolare la direttiva `pragma omp parallel for schedule(SCHEDULE_TYPE)`. Per il parametro `SCHEDULE_TYPE` è stato utilizzato `guided`. In particolare nel caso in cui le righe avessero un numero variabile di elementi non nulli, parametri `col_index` e `data`, allora alcuni thread potrebbero dover elaborare più operazioni di altri e in questo modo si assicura che le righe che hanno più elementi non nulli vengano processate per prima lasciando quelle più leggere per dopo, bilanciando il carico dinamicamente. Inoltre, utilizzando gli array monodimensionali per la memorizzazione dei dati, i thread possono accedere agli elementi di `col_index` e di `data` in maniera più prevedibile riducendo i salti di memoria e favorendo il prefetching della CPU. Dato che ogni thread processa una riga in maniera indipendente, ovvero utilizza `offsets` per sapere quali elementi leggere in `col_index` e `data`, non ci sono aggiornamenti concorrenti sugli stessi dati in modo da evitare il problema del false cache sharing; in questo modo si cerca di non avere situazioni in cui si abbiano dei thread inattivi in quei casi in cui si abbiano degli squilibri significativi nelle dimensioni delle righe.
L'algoritmo per il calcolo dell'operazione di prodotto matrice-vettore utilizzando il formato `HLL` può essere rappresentato con il seguente algoritmo:

Algorithm 5 hll_parallel_product(hll_matrix, x, y, num_threads, time_used)

#*hll_matrix*: matrice nel formato HLL
#x: Vettore di input
#y: Vettore di output
#*num_threads*: Numero di thread da utilizzare
#*time_used*: Tempo di esecuzione misurato

$rows \leftarrow num\_of\_rows(hll\_matrix, 0)$
$start \leftarrow$ tempo corrente

Parallelizzazione: Parallel for con `num_threads`, strategia `guided`

for $h = 0$ **to** $hll\_matrix.hacks\_num - 1$ **do**
  $rows\_in\_h \leftarrow num\_of\_rows(hll\_matrix, h)$
  $max\_nzr\_h \leftarrow hll\_matrix.max\_nzr[h]$
  $offset\_h \leftarrow hll\_matrix.offsets[h]$
  for $r = 0$ **to** $rows\_in\_h - 1$ **do**
    $sum \leftarrow 0.0$
    $data\_ptr \leftarrow hll\_matrix.data[offset\_h + r * max\_nzr\_h]$
    $col\_ptr \leftarrow hll\_matrix.col\_index[offset\_h + r * max\_nzr\_h]$
SIMD per il prodotto scalare con prefetching
    for $j = 0$ **to** $max\_nzr\_h - 1$ **do**
      Prefetch di $data\_ptr[j + 4]$ e $col\_ptr[j + 4]$
      $sum \leftarrow sum + data\_ptr[j] * x[col\_ptr[j]]$
    end for
    $y[rows * h + r] \leftarrow sum$
  end for
end for
$end \leftarrow$ tempo corrente $*time\_used \leftarrow end - start$

In combinazione con la direttiva di bilanciamento del carico vengono utilizzate le direttive `default(none)` e `shared(hll_matrix, x, y, rows)`, dove:

- `default(none)`: impone di dichiarare esplicitamente quali variabili sono condivise e quali private, evitando errori dovuti a variabili non inizializzate correttamente;
- `shared(hll_matrix, x, y, rows)`: vengono indicate quali variabili sono condivise tra i thread utilizzati per il calcolo evitando di dover fare delle copie superflue e di avere un accesso alle stesse istanze di dati senza aver bisogno di sincronizzazione avendo però il rischio di avere situazioni di *race condition*, ovvero se più thread dovessero scrivere su una variabile condivisa senza sincronizzazione si potrebbero avere risultati non deterministici. Considerando le variabili indicate:
  - `hll_matrix`: la variabile rappresenta la rappresentazione della matrice nel formato utilizzato e viene solo letta dai vari thread e può essere condivisa senza problemi senza avere problemi di race condition. Tutti i thread accedono a porzioni differenti della matrice e nessun thread modifica la matrice e quindi può essere condivisa senza problemi;
  - `x`: rappresenta il vettore moltiplicato per la matrice, tutti i thread lo leggono e nessuno lo modifica, dato che il suo accesso è read-only può essere condiviso senza memoria;
  - `y`: indica il vettore risultate, ma ogni thread calcola una riga diversa di `y`, quindi non ci sono conflitti sulle righe se ogni riga è scritta una sola volta. Si avrebbe una

condizione di race condition se più thread cercassero di scrivere contemporaneamente sullo stesso `y[i]`, quindi, il vettore risultate può essere condiviso senza problemi.

- `rows`: è una variabile che contiene il numero di righe della matrice, il suo valore è costante e non cambia mai, i thread lo utilizzano per capire fino a che punto devono iterare e quindi è possibile condividerlo senza problemi.

In questo modo si evitano copie inutili della matrice matrice e dei vettori, si garantisce un accesso efficiente alla memoria e non si introducono problemi di concorrenza dato che le variabile vengono utilizzate in modo sicuro. Per il calcolo effettivo dei coefficienti del vettore risultante viene utilizzata la direttiva OpenMP `#pragma omp simd reduction(+ : sum)` indicando che il ciclo `for` viene vettorizzato automaticamente dal compilatore, sfruttando le istruzioni SIMD (Single Instruction, Multiple Data), vengono evitati problemi di race condition accumulando i risulati in modo sicuro.
Come nel caso del calcolo parallelo utilizzando il formato `CSR`, il tempo viene misurato utilizzando la funzione `omp_get_wtime()`.
Viene utilizzato il prefetching esplicito in modo da ridurre la latenza della memoria caricando in anticipo in cache, in particolare indicando la compilatore di caricare i dati prima che siano necessari in modo da migliorare l'efficienza delle letture.
La matrice è divisa in blocchi chiamati "hack", ognuno contenente un sottoinsieme di righe. La variabile `hacks_num` indica il numero totale di hack. Ogni thread elabora uno o più hack in parallelo, questo perché il numero di hack potrebbe essere maggiore del numero di thread utilizzati, indicato dalla variabile `num_threads`.
Nell'implementazione viene utilizzata la fuzione `__builtin_prefetch()` per effettuare il prefetch esplicito dei dati nella cache prima che vengano effettivamente utilizzati, con l'obiettivo di ridurre la latenza di accesso alla memoria e migliorare le prestazioni.

8 CUDA

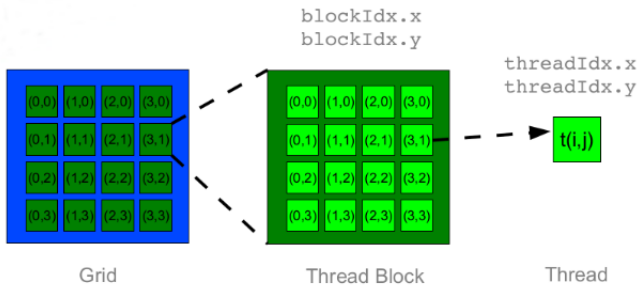
8.1 Introduzione

CUDA è una piattaforma di calcolo parallelo e un modello di programmazione creato da NVIDIA per sfruttare le GPU nelle operazioni di calcolo generico con cui è possibile accelerare notevolmente le applicazioni di calcolo utilizzando la potenza di elaborazione delle GPU.
CUDA permette agli sviluppatori di scrivere codice in linguaggi come C, C++, Fortran, Python e MATLAB, utilizzando semplici estensioni del linguaggio per definire il parallelismo in modo intuitivo ed efficiente.
Nei programmi che sfruttano l'accelerazione della GPU, la parte di codice sequenziale viene eseguita sulla CPU, ottimizzata per operazioni single-thread, mentre i calcoli più complessi e intensivi vengono distribuiti sui core della GPU in esecuzione parallela.
L'architettura di CUDA è basata su un'organizzazione gerarchica di thread ben definita utilizzata con l'obiettivo di massimizzare l'efficienza nell'elaborazione parallela.
I thread sono raggruppati in blocchi, i blocchi sono organizzati in una griglia. Ogni thread e ogni blocco sono identificati da indici univoci, che possono essere utilizzati per accedere in modo indipendente ai dati.
L'organizzazione dei thread in blocchi, e nella relativa griglia, può essere multidimensionale: i thread all'interno di un blocco e i blocchi all'interno della griglia possono essere organizzati in una, due o tre dimensioni:

- se la griglia o il blocco sono monodimensionali si avrà un solo indice, `x`, per identificare un thread o un blocco;
- se la griglia o il blocco sono bidimensionali si utilizzeranno due indici per l'indirizzamento, `x` e `y`;

- se la griglia o il blocco sono tridimensionali allora si utilizzeranno tre indici, `x`, `y` e `z` per identificare un thread o un blocco.

La strutturazione consente ai thread di accedere in maniera indipendente a porzioni di dati differenti, in modo da poter organizzare e getire i dati in memoria in maniera efficiente.
I thread vengono eseguiti da più Streaming Multiprocessor sulla GPU, ognuno dei quali ha la responsabilità di gestione ed esecuzione di warp. Uno warp, nel'architetture NVIDIA, identifica un gruppo di 32 thread consecutivi.
La distribuzione dei therad sugli Streaming Multiprocessor deve essere bilanciata per evitare situazioni di sovraccarico o sottoutilizzo della GPU. Per questo motivo la scelta della dimensione della griglia in CUDA influisce in modo diretto sulle prestazioni dato che determina come i blocchi di thread vengono assegnati ai multiprocessori della GPU.
Di seguito viene rappresentato il modo utilizzato dall'architettura per indicizzare i thread:



L'effettiva indicizzazione di un thread può avvenire nel seguente modo:

```
i:  blockIdx.x * blockDim.x + threadIdx.x
j:  blockIdx.y * blockDim.y + threadIdx.y
```

Bisogna considerare che bisogna trovare dimensione e numero di blocchi ottimale considerando che:

- la dimensione del blocco dovrebbe essere multiplo della grandezza degli Streaming Multiprocessor dato che i thread nello warp vengono eseguiti in un unico ciclodi clock. Scegliere una dimensione differente porta a considerazioni differenti:
  - scegliere una dimensione del blocco con meno di 32 thread comporta alla creazione di un solo warp di cui solo alcuni sono attivi, mentre gli altri sono inattivi;
  - utilizzare multipli di 32 thread e vengono creati tanti warp per coprire tutti i thread che vengono schedulati ed eseguiti indipendentemente dallo Streming Multiprocessor;
  - scegliere di allocare molti thread con una dimensione non multipla di 32, questo comporta che l'ultimo warp non è completamente utilizzato.
- non è possibile superare il limite fisico dell'hardware sottostante, che nella GPU utilizzata per raccogliere i dati supporta blocchi di dimensione massima di 1024 thread.

Un altro aspetto da considerare è il caso in cui il numero di blocchi considerato supera il numero di Streaming Multiprocessor disponibili, in questo caso si entra nella gestione della concorrenza in CUDA. In questo caso i blocchi non vengono eseguiti contemporaneamente, ma vengono programmati dinamicamente sugli Streaming Multiprocessor man mano che le risorse diventano disponibili e questo continua fintanto che non ci sono blocchi da completare.
Le funzioni che vengono eseguite sulla GPU, i `kernel`, utilizzano anche un altro fattore importante che può influenzare le prestazioni delle operazioni, ovvero il tipo di memoria utilizzato. In particolare, per il prodotto matrice sparsa-vettore, sono stati utilizzati i seguenti tipi di memoria:

- **memoria globale**: è la memoria principale accessibile da tutti i thread di tutti i blocchi, è dotata di una grande dimensione ma l'accesso a questo tipo di memoria è costoso in termini di latenza. Per poter diminuire la latenza di accesso si utilizzano gli accessi coalescenti;

- **memoria condivisa**: è una memoria più veloce della memoria globale ed è accessibile solo ai thread dello stesso blocco, è utile per una cooperazione tra i thread e ridurre gli accessi alla memoria globale. Ogni blocco ha una porzione separata di shared memory. In generale ha una dimensione ridotta rispetto alla memoria globale. Vengono ridotti gli accessi in memoria globale;

- **cache L2**: è una cache di secondo livello condivisa da tutti gli Streaming Multiprocessor ed è una memoria intermedia tra la memoria globale e registri/shared memory. Utilizzando questo tipo di memoria si migliorano le prestazioni memorizzando i dati più utilizzati riducendo gli accessi ripetuti alla memoria globale.
  La gestione è automatica dall'hardware della GPU, il programmatore non può, e non deve, imporre cosa mettere o togliere in cache.
  Il contenuto è basato sugli accessi precedenti, se si accede spesso agli stessi dati allora la cache L2 manterrà quei dati. Nello studio in esame viene consultata la cache L2 per verificare se un dato è presenta o no nella cache.
  Nel caso in cui il dato è presente allora viene tornata altrimenti il dato viene preso dalla memoria globale, restituito al programma per poi essere caricato in cache L2 per gli accessi futuri, in questo caso in eventuali accessi futuri allo stesso dato si avranno benefici in termini di latenza.
  Il suo accesso, quindi, è automatico e più veloce rispetto alla memoria globale ed è ottima per accessi locali e ripetuti per la stessa porzione di dati.

Con accesso coalescente si intende un tipo di accesso in cui i thread accedono ad indirizzi di memoria contigui, in questo modo è possibile riunire gli accessi in un'unica transazione di memoria riducendo drasticamente il numero di accessi e la conseguente latenza.
Una cosa da considerare è che la matrice considerata per il prodotto, sia quella in formato `CSR` che in formato `HLL`, che il vettore utilizzato per il prodotto che il vettore risultante vengono copiate dal sistema host al device, dove con sistema host si intende la CPU e la memoria mentre con device si intende la GPU con la sua memoria dedicata.
Le operazioni coinvolte per il processamento di un'applicazione parallela sono:

- allocazione della memoria sul device, utilizzando l'api `cudaMalloc`;

- copia dei dati dalla memoria dal sistema host al device, si utilizza l'api `cudaMemcpy` utilizzando il flag `cudaMemcpyHostToDevice`;

- lanciare il kernel con il codice per il calcolo del prodotto matrice-sparsa vettore, invocazione simile a `myKernel<<<gridSize, blockSize>>>`, dove `myKernel` indica il nome della funzione che implementa il kernel che verrà eseguito sul device;

- copiare il risultato dalla memoria del device alla memoria dell'host, anche in questo caso si utilizza l'api `cudaMemcpy` ma si utilizza il flag `cudaMemcpyDeviceToHost`;

- liberazione della memoria allocata sul device, per questa operazione si utilizza l'api `cudaFree`.

Per entrambi i formati di memorizzazione delle matrici, sono stati eseguiti i differenti kernel in diverse configurazioni, in particolare vengono utilizzati un numero differente di thread per l'esecuzione. Il numero di thread utilizzati è rappresentato dai coefficienti del seguente vettore:

$$\texttt{threads\_number[5]} = \{32, 64, 96, 128, 160\}$$

In questo modo i kernel vengono lanciati utilizzando tutti i thread che compongono uno warp, rispettivamente ogni kernel viene lanciato utilizzando rispettivamente 1, 2, 3, 4 e 5 warp per blocco. In questo modo si utilizzano tutti i thread appartenenti ad uno warp in modo da non avere sottoutilizzi. Per entrambi i formati non è stato utilizzato un approccio in cui si cercasse di saturare completamente le risorse della GPU ma si è utilizzato un approccio conservativo.

8.2 CSR

Per parallelizzare il prodotto matrice-vettore con una matrice sparsa in formato CSR, sono state sviluppate quattro varianti.

8.2.1 csr_matvec_kernel

Questo kernel rappresenta una prima soluzione, scelta per la sua semplicità concettuale e di implementazione. Nonostante la sua struttura basilare, riesce comunque a garantire prestazioni soddisfacenti. La dimensione della griglia dei blocchi viene determinata dalla formula:

$$\texttt{grid\_dim1} = \frac{M + \texttt{num\_threads\_per\_block} - 1}{\texttt{num\_threads\_per\_block}}$$

dove `num_threads_per_block` corrisponde all'i-esimo valore del vettore `threads_number`. Ciascun thread, in questo metodo, è incaricato di elaborare una singola riga della matrice. L'assegnazione delle righe avviene tramite il calcolo di un indice globale univoco, che identifica la riga su cui il thread dovrà operare. Dopo aver determinato la riga corrispondente, il thread procede con l'esecuzione del prodotto. Di seguito, l'algoritmo:

Algorithm 6 csr_matvec_kernel(d_IRP, d_JA, d_AS, rows, d_x, d_y)

```
#d_IRP: Vettore IRP della matrice CSR
#d_JA: Vettore JA della matrice CSR
#d_AS: Vettore d_AS della matrice CSR
#rows: numero di righe della matrice CSR
#d_x: Vettore che effettua il prodotto
#d_y: Vettore risultato del prodotto
row ← blockIdx.x × blockDim.x + threadIdx.x
if row < rows then
    sum ← 0.0
    start ← d_IRP[row]
    end ← d_IRP[row + 1]
    for i ← start to end − 1 do
        col ← [d_JA[i]]
        val ← d_AS[i]
        sum ← sum + val × col
    end for
    d_y[row] ← sum
end if
```

8.2.2 csr_matvec_shfl_reduction

In questa seconda soluzione, ogni warp (composto da 32 thread) è responsabile del calcolo di una singola riga della matrice. Ogni blocco CUDA è costituito da N warp, dove N dipende

dall'elemento selezionato nel vettore `threads_number`. La dimensione della griglia viene calcolata dinamicamente in base al numero di righe M della matrice, secondo la seguente formula:

$$\texttt{grid\_dim2.x} = \frac{M + \texttt{blockDim2.y} - 1}{\texttt{blockDim2.y}}$$

dove `blockDim2.y` corrisponde esattamente all'i-esimo valore del vettore `threads_number` che viene considerato. Inoltre ogni thread di un warp calcola una parte del prodotto scalare tra una riga della matrice sparsa e il vettore x, dunque il risultato finale di ciascun thread rappresenta solo una somma parziale. Di seguito lo sviluppo del kernel:

Algorithm 7 csr_matvec_shfl_reduction(d_mat, M, x, y)

#*d_mat*: Matrice CSR
#M: numero di righe della matrice CSR
#x: Vettore che effettua il prodotto
#y: Vettore risultato del prodotto

$row \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$

$lane \leftarrow threadIdx.x$

if $row < M$ **then**

  $sum \leftarrow 0.0$

  $start \leftarrow d\_mat.IRP[row]$

  $end \leftarrow d\_mat.IRP[row + 1]$

Ogni thread processa elementi della riga

  for $j \leftarrow start$ **to** $end - 1$ **do**

    $col \leftarrow [d\_mat.JA[j]]$

    $val \leftarrow d\_mat.AS[j]$

    $sum \leftarrow sum + val \times col$

    $j \leftarrow j + WARP\_SIZE$

  end for

Riduzione warp-level usando __shfl_sync

  for $offset \leftarrow WARP\_SIZE$ **to** 0 **do**

    $sum \leftarrow sum + \_\_shfl\_sync(0xFFFFFFFF, sum, lane - offset)$

    $offset \leftarrow offset/2$

  end for

Solo il primo thread del warp scrive il risultato finale

  if $lane == 0$ **then**

    $y[row] \leftarrow sum$

  end if

end if

Poiché ogni riga della matrice deve essere elaborata in parallelo da più thread di un warp, è necessario combinare i risultati parziali in un unico valore.
Questo processo è chiamato `riduzione`, e permette di ottenere il valore finale da scrivere nel vettore di output. L'implementazione di questa funzionalità può essere fatta con la funzione `__shfl_sync` e consente la comunicazione diretta tra i thread di uno stesso warp, riducendo la necessità di sincronizzare la memoria globale. Tuttavia, con variabili di tipo double, l'uso di questa funzione può introdurre piccoli errori di accumulo a causa della limitata precisione numerica.

Le operazioni di somma tra numeri in virgola mobile non sono perfettamente associative, cioè:

$$\texttt{(a+b)+c} \neq \texttt{a+(b+c)}$$

8.2.3 csr_matvec_shared_memory

Questo kernel ha una struttura simile a quella del precedente, ma introduce un concetto fondamentale: la `shared memory`. In particolare, sfrutta l'esecuzione parallela sulla GPU suddividendo il calcolo tra i warp dei thread, e utilizza la `shared memory` per ridurre la pressione sulla memoria globale e ottimizzare l'accumulo dei risultati parziali. La dimensione della griglia viene calcolata dinamicamente in base al numero di righe M della matrice, secondo la seguente formula:

$$\texttt{grid\_dim3.x} = \frac{M + \texttt{blockDim3.y} - 1}{\texttt{blockDim3.y}}$$

dove `blockDim3.y` corrisponde esattamente all'i-esimo valore del vettore `threads_number` che viene considerato. Tuttavia, è importante osservare che l'uso della `shared memory` non migliora direttamente la coalescenza degli accessi alla memoria globale. Gli accessi risultano infatti solo parzialmente coalescenti, a causa della natura sparsa dei dati nella matrice in formato `CSR`. Di conseguenza, all'interno di un warp, alcuni thread accedono a posizioni contigue e coalescenti, mentre altri no. Ogni thread della GPU è responsabile del calcolo di una parte del prodotto scalare tra una riga della matrice e il vettore di input. In particolare, i thread all'interno di uno warp collaborano per calcolare la somma parziale degli elementi della riga, sfruttando la shared memory per memorizzare i risultati temporanei. Il kernel dunque si basa su quattro fasi principali:

- **Identificazione della riga da elaborare**:
  Ogni thread determina la riga della matrice da elaborare utilizzando gli indici di blocco, mentre la sua posizione nello warp viene calcolata per facilitare la fase di riduzione.

- **Calcolo del prodotto scalare**:
  I thread appartenenti a un warp suddividono il carico di lavoro e calcolano parzialmente il prodotto scalare tra la riga della matrice (`d_csr.AS`) e il vettore di input (`d_x`). Per garantire un'efficiente distribuzione del carico, ogni thread processa elementi a intervalli regolari, avanzando di `WARP_SIZE` posizioni per volta.

- **Utilizzo della memoria condivisa e riduzione**:
  I risultati parziali calcolati da ciascun thread vengono memorizzati nella memoria condivisa (`shared_data`). Successivamente, viene eseguita una riduzione parallela all'interno del warp, sommando progressivamente i contributi parziali fino a ottenere il valore finale del prodotto scalare. Per coordinare i thread durante questa operazione, vengono utilizzate le barriere di sincronizzazione (`__syncthreads()`), che impediscono letture/scritture premature. La riduzione è stata suddivisa in passaggi successivi, iniziando con una somma a 16, perché `WARP_SIZE` è 32.

- **Salvataggio del risultato finale**:
  Il thread leader (`lane` $== 0$) salva il risultato finale nella memoria globale, completando l'elaborazione della riga nel vettore di output.

Algoritmo indicato a pagina seguente.

Algorithm 8 csr_matvec_shared_memory(d_mat, M, x, y)

#d_mat: Matrice CSR
#M: numero di righe della matrice CSR
#x: Vettore che effettua il prodotto
#y: Vettore risultato del prodotto

$row \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$

$lane \leftarrow threadIdx.x$

if $row < M$ **then**

$sum \leftarrow 0.0$

$start \leftarrow d\_mat.IRP[row]$

$end \leftarrow d\_mat.IRP[row + 1]$

for $j \leftarrow start$ **to** $end - 1$ **do**

$col \leftarrow [d\_mat.JA[j]]$

$val \leftarrow d\_mat.AS[j]$

$sum \leftarrow sum + val \times col$

$j \leftarrow j + WARP\_SIZE$

end for

Scrittura risultato parziale all'interno della memoria condivisa

$shared\_data[lane + threadIdx.y * WARP\_SIZE] \leftarrow sum$

$\_\_syncthreads()$

Riduzione coalescente

if $lane < 16$ **then**

$shared\_data[lane+threadIdx.y*WARP\_SIZE] \mathrel{+}= shared\_data[lane+threadIdx.y*WARP\_SIZE+16]$

$\_\_syncthreads()$

end if

if $lane < 8$ **then**

$shared\_data[lane+threadIdx.y*WARP\_SIZE] \mathrel{+}= shared\_data[lane+threadIdx.y*WARP\_SIZE+8]$

$\_\_syncthreads()$

end if

if $lane < 4$ **then**

$shared\_data[lane+threadIdx.y*WARP\_SIZE] \mathrel{+}= shared\_data[lane+threadIdx.y*WARP\_SIZE+4]$

$\_\_syncthreads()$

end if

if $lane < 2$ **then**

$shared\_data[lane+threadIdx.y*WARP\_SIZE] \mathrel{+}= shared\_data[lane+threadIdx.y*WARP\_SIZE+2]$

$\_\_syncthreads()$

end if

if $lane < 1$ **then**

$shared\_data[lane+threadIdx.y*WARP\_SIZE] \mathrel{+}= shared\_data[lane+threadIdx.y*WARP\_SIZE+1]$

$\_\_syncthreads()$

end if

if $lane == 0$ **then**

$y[row] \leftarrow shared\_data[threadIdx.y * WARP\_SIZE]$

end if

end if

8.2.4 csr_matvec_warp_cacheL2

Il quarto kernel esegue l'operazione SpMV in formato CSR, implementando ottimizzazioni avanzate per migliorare l'efficienza computazionale. La dimensione della griglia viene calcolata dinamicamente in base al numero di righe M della matrice, secondo la seguente formula:

$$\texttt{grid\_dim4.x} = \frac{M + \texttt{blockDim4.y} - 1}{\texttt{blockDim4.y}}$$

dove `blockDim4.y` corrisponde esattamente all'i-esimo valore del vettore `threads_number` che viene considerato. Le principali tecniche adottate sono:

- Caching L2 con `__ldg()`: utilizza la read-only memory cache per velocizzare gli accessi ai dati della matrice e del vettore, riducendo la latenza della memoria globale.
- Riduzione warp-wide con `__shfl_sync()`: esegue la somma parziale dei prodotti direttamente tra thread dello stesso warp, evitando l'uso della shared memory e riducendo il numero di accessi alla memoria globale.

Algorithm 9 csr_matvec_warp_cacheL2(d_mat, M, x, y)

#*d_mat*: Matrice CSR
#M: numero di righe della matrice CSR
#x: Vettore che effettua il prodotto
#y: Vettore risultato del prodotto

$row \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$

$lane \leftarrow threadIdx.x$

if $row < M$ **then**

  $sum \leftarrow 0.0$

  $start \leftarrow d\_mat.IRP[row]$

  $end \leftarrow d\_mat.IRP[row + 1]$

  $*AS\_ptr \leftarrow d\_Mat.AS + start$

  $*JA\_ptr \leftarrow d\_Mat.JA + start$

  for $j \leftarrow lane$ **to** $end - start$ **do**

    $a\_val \leftarrow \_\_ldg(\&JA\_ptr[j])$

    $col\_index \leftarrow \_\_ldg(\&JA\_ptr[j])$

    $sum \leftarrow sum + a\_val \times \_\_ldg(\&x[col\_index])$

    $j \leftarrow j + WARP\_SIZE$

  end for

Riduzione warp-level usando __shfl_sync

  for $offset \leftarrow WARP\_SIZE/2$ **to** 0 **do**

    $sum \leftarrow sum + \_\_shfl\_sync(0xFFFFFFFF, sum, lane - offset)$

    $offset \leftarrow offset/2$

  end for

Solo il primo thread del warp scrive il risultato finale

  if $lane == 0$ **then**

    $y[row] \leftarrow sum$

  end if

end if

8.3 HLL

8.3.1 cuda_hll_kernel_v1

La prima implementazione del nucleo di calcolo è abbastanza semplice, utilizzato sia per la semplicità concettuale che per la facilità implementativa. La dimensione della griglia è calcolata nel seguente modo:

$$\texttt{grid\_dim} = \frac{M + \texttt{num\_threads\_per\_block} - 1}{\texttt{num\_threads\_per\_block}}$$

Dove il valore di `num_threads_per_block` corrisponde ad uno dei coefficienti all'interno del vettore `threads_number`.
Di seguito la rappresentazione dell'algoritmo:

Algorithm 10 `cuda_hll_kernel_v1(hack_size, hacks_num, data, offsets, col_index, max_nzr, M, x, y)`

#hack_size: dimensione di un hack
#hack_num: numero di hack della matrice
#data: array indicante i valori non nulli della matrice
#offsets: array per individuare le righe di inizio e fine blocco
#col_indexes: array contenente gli indici delle colonne dei valori non nulli
#max_nzr: array contente il numero di non zeri per blocco
#M: numero di righe e colonne della matrice
#x: vettore moltiplicativo
#y: vettore risultante

$index \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$

if $index \geq hacks\_num$ **then**

  return

end if

$start\_row \leftarrow index \times hack\_size$

$end\_row \leftarrow min(start\_row + hack\_sirowze, M)$

$max\_nzr\_h \leftarrow max\_nzr[index]$

$base\_offset \leftarrow offsets[index]$

for $row\_index \leftarrow start\_row$ **to** end_row **do**

  $sum \leftarrow 0.0$

  $row\_offsets \leftarrow base\_offset + (row\_index - start\_row) \times max\_nzr\_h$

  for $j \leftarrow 0$ **to** max_nzr_h **do**

    $sum+ = data[row\_offsets + j] \times x[col\_index[row\_offsets + j]]$

  end for

  $y[col\_index] \leftarrow sum$

end for

Nell'implementazione il tipo di memoria utilizzata è quella globale, ovvero il tipo di memoria più grande ma con le latenze di accesso più grandi.
Si utilizza un thread per hack, in particolare ogni thread calcola il proprio indice, calcola il proprio intervallo di righe per quell'hack per poi accedere ai coefficienti delle righe e agli elementi del vettore moltiplicativo per poi scrivere il risultato del calcolo nel vettore risultante.
La riduzione avviene in modo implicito. Il risultato della moltiplicazione tra i coefficienti della matrice e del vettore viene accumulato in una variabile.
Una volta che le iterazioni sono terminate, il valore della variabile che ha accumulato i calcoli fatti finora viene copiato nel vettore risultate.
Dato che si utilizza la memoria globale, gli accessi non sono coalescenti.

8.3.2 cuda_hll_kernel_v2

Per l'implementazione del secondo nucleo di calcolo in CUDA si deciso di assegnare una riga per thread.
La dimensione della griglia è calcolata nel seguente modo:

$$\texttt{grid\_dim} = \frac{M + \texttt{block\_dim.y} - 1}{\texttt{block\_dim.y}}$$

Dove il valore `block_dim.y` rappresenta il numero di warp per blocco, che varia in base alla configurazione con cui si lancia l'esecuzione del kernel.
Di seguito la rappresentazione dell'algoritmo:

Algorithm 11 `cuda_hll_kernel_v2(hack_size, hacks_num, data, offsets, col_index, max_nzr, M, x, y)`

#hack_size: dimensione di un hack
#hack_num: numero di hack della matrice
#data: array indicante i valori non nulli della matrice
#offsets: array per individuare le righe di inizio e fine blocco
#col_indexes: array contenente gli indici delle colonne dei valori non nulli
#max_nzr: array contente il numero di non zeri per blocco
#M: numero di righe e colonne della matrice
#x: vettore moltiplicativo
#y: vettore risultante

$index \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$

if $index \geq M$ **then**

  return

end if

$hack \leftarrow index/hack\_size$

$local\_row \leftarrow index\%hack\_size$

$row\_start \leftarrow offsets[hack] + local\_row \times max\_nzr[hack]$

$row\_end \leftarrow row\_start + max\_nzr[hack]$

$sum \leftarrow 0.0$

for $j \leftarrow row\_start$ **to** row_end **do**

  $sum+ = data[j] \times x[col\_index[j]]$

end for

$y[col\_index] \leftarrow sum$

Anche in questo caso si utilizza la memoria globale.
Ogni thread calcola il proprio indice, l'hack su cui andare a lavorare e l'indice della riga che gli è stata assegnata per poi calcolare l'intervallo degli indici che utilizzerà per considerare gli elementi non nulli della matrice considerata.
Anche in questo caso, dato l'utilizzo della memoria globale, gli accessi in memoria non sono coalescenti.

8.3.3 cuda_hll_kernel_v3

Per l'implementazione del terzo kernel si è deciso di utilizzare la memoria condivisa per diminuire la latenza degli accessi in memoria, utilizzando sempre un thread per riga.
La dimensione della griglia è calcolata come nel caso precedente.
Per completezza si ripete la formula:

$$\texttt{grid\_dim} = \frac{M + \texttt{block\_dim.y} - 1}{\texttt{block\_dim.y}}$$

Dove il valore `block_dim.y` rappresenta il numero di warp per blocco, che varia in base alla configurazione con cui si lancia l'esecuzione del kernel.
Di seguito la rappresentazione dell'algoritmo:

Algorithm 12 `cuda_hll_kernel_v3(hack_size, hacks_num, data, offsets, col_index, max_nzr, M, x, y)`

#hack_size: dimensione di un hack
#hack_num: numero di hack della matrice
#data: array indicante i valori non nulli della matrice
#offsets: array per individuare le righe di inizio e fine blocco
#col_indexes: array contenente gli indici delle colonne dei valori non nulli
#max_nzr: array contente il numero di non zeri per blocco
#M: numero di righe e colonne della matrice
#x: vettore moltiplicativo
#y: vettore risultante

$index \leftarrow threadIdx.x$

$global\_index \leftarrow blockIdx.x \times blockDim.x + index$

if $index < M$ **then**

  $shared\_x[index] \leftarrow x[index]$

end if

$\_\_syncthreads()$

if $global\_index < M$ **then**

  $hack \leftarrow global\_index / hack\_size$

  $local\_offset \leftarrow (global\_index \% hack\_size) \times max\_nzr[hack] + offsets[hack]$

  $acc \leftarrow 0.0$

  for $j \leftarrow 0$ **to** $max\_nzr[hack]$ **do**

    $col \leftarrow col\_index[local\_offset + j]$

    $value \leftarrow data[local\_offset + j]$

    if $col < hack\_size$ **then**

      $acc += value \times shared\_x[col]$

    else

      $acc += value \times x[col]$

    end if

  end for

  $y[global\_index] \leftarrow acc$

end if

Per prima cosa si caricano i dati del vettore `x` nella memoria condivisa, `shared_x`, ogni thread carica un elemento nella memoria condivisa se il suo indice è minore del valore delle righe e colonne. Con `__syncthreads()` si garantisce che tutti i trhead abbiano caricato i dati nella memoria condivisa prima che proseguano evitando di avere condizioni di competizione.
La logica del calcolo è simile al caso del kernel precedente, l'unica differenza è sull'accesso del coefficiente del vettore moltiplicativo. In particolare se l'indice di colonna è minore della dimensione dell'hack, ovvero `hack_size`, si accede al dato dalla memoria condivisa altrimenti in memoria globale.
Il valore di `hack_size` utilizzato è sempre pari a 32, pari al numero di thread all'interno di uno warp. Una volta ottenuto il valore dell'indice di colonna si calcola l'effettivo risultante.
Per quanto riguarda gli accessi in memoria, quando si accede alla memoria condivisa gli accessi sono coalescenti dato che tutti i thread di un blocco accedono alla memoria condivisa in modo lineare.

Rispetto alla due implementazioni precedenti si sfrutta proprio la capacità della memoria condivisa per sfruttare la coalescenza degli accessi ai dati.
Nel caso in cui si accede alla memoria globale allora gli accessi potrebbero non essere coalescenti, questo perché i thread potrebbero accedere a porzioni di memoria non contigue.
Per quanto riguarda la riduzione, ogni thread effettua la propria riduzione locale per un elemento del vettore risultate e non ci sono riduzioni parallele o globali tra thread, quindi ogni thread lavora in modo indipendente.

8.3.4 cuda_hll_kernel_v4

Per l'implementazione dell'ultimo kernel si è deciso di utilizzare un warp per riga e di utilizzare una riduzione intra-warp, ovvero una somma parziale tra i thread dello stesso warp, in modo efficiente.
La dimensione della griglia è calcolata nel seguente modo:

$$\texttt{grid\_dim} = \frac{M \times 32}{\texttt{thread\_used}}$$

Dove il valore `M` rappresenta il numero di righe e colonne, mentre il valore `thread_used` indica il numero di thread utilizzato.
Si è deciso di utilizzare la precedente formula per avere la garanzia di poter avere uno warp per riga, cercando di avere una distribuzione del carico di lavoro in modo bilanciato tra i blocchi.
Nella prima fase dell'algoritmo viene calcolato l'id del thread, l'id del warp in cui si trova il thread e l'indice del del thread all'interno del warp. Il valore della riga da considerare viene assegnato allo stesso valore dello warp.
Successivamente vengono calcolati gli indici `row_start` e `row_end` per sapere su quali elementi della riga leggere i valori non nulli.
Ogni thread del warp elabora una parte della riga, per poi saltare di 32 elementi in modo tale che ogni thread all'interno del warp contribuisca a calcolare il prodotto per una singola riga.
La funzione `__shfl_down_sync` consente una riduzione warp-level efficiente spostando valori tra thread all'interno dello stesso warp. È utilizzata per combinare i risultati parziali dei thread di un warp senza utilizzare memoria condivisa, in moda da ridurre le latenze di accesso e la necessità di sincronizzazioni esplicite.
Successivamente, il thread di indice 0 all'interno del warp scrive il risultato del prodotto nella posizione corretta nel vettore risultante.
Per quanto riguarda gli accessi, i thread di uno warp accedono agli elementi di `data` e di `col_index` in modo consecutivo e allineato, quindi possono essere considerati come accessi coalescenti.
Discorso che non vale con l'accesso al vettore moltiplicativo `x`, in quanto si utilizza un accesso indiretto dove i valori dipendono da `col_index[element]` e non necessariamente è coalescente perché i valori di `col_index` potrebbero puntare a posizioni arbitrarie di `x` e quindi avere accessi irregolari.
Bisogna sottolineare che in questa soluzione è stata utilizzata la memoria globale per l'accesso agli array di interesse, e ogni thread accede alle strutture per leggere e solo quelli con indice di warp pari a 0 accede all'elemento corretto del vettore risultante.
Prima di proseguire con l'algoritmo, è opportuno considerare che il calcolo di alcuni indici viene fatto utilizzando gli operatori bitwise del `C` cercando di ottimizzare il calcolo di alcuni indici di interesse.
Di seguito la rappresentazione dell'algoritmo:

Algorithm 13 `cuda_hll_kernel_v4(hack_size, hacks_num, data, offsets, col_index, max_nzr, M, x, y)`

#hack_size: dimensione di un hack
#hack_num: numero di hack della matrice
#data: array indicante i valori non nulli della matrice
#offsets: array per individuare le righe di inizio e fine blocco
#col_indexes: array contenente gli indici delle colonne dei valori non nulli
#max_nzr: array contente il numero di non zeri per blocco
#M: numero di righe e colonne della matrice
#x: vettore moltiplicativo
#y: vettore risultante

$thread\_id \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$

$warp\_id \leftarrow thread\_id >> 5$

$lane \leftarrow thread\_id \ \& \ 31$

$row \leftarrow warp\_id$

if $row < M$ **then**

  $hack \leftarrow row/hack\_size$

  $row\_start \leftarrow (row\%hack\_size) \times max\_nzr[hack] + offset[hack]$

  $row\_end \leftarrow row\_start + max\_nzr[hack]$

  $sum \leftarrow 0.0$

  for $element \leftarrow row\_start + lane$ **to** row_end **do**

    $sum+ = data[element] \times x[col\_index[element]]$

  end for

  for $offset \leftarrow warpSize >> 1$ **to** $offsets > 0$ **do**

    $sum += \ \_\_shfl\_down\_sync(0xFFFFFFFF, sum, offset)$

  end for

  if $lane == 0$ **then**

    $y[row] \leftarrow sum$

  end if

end if

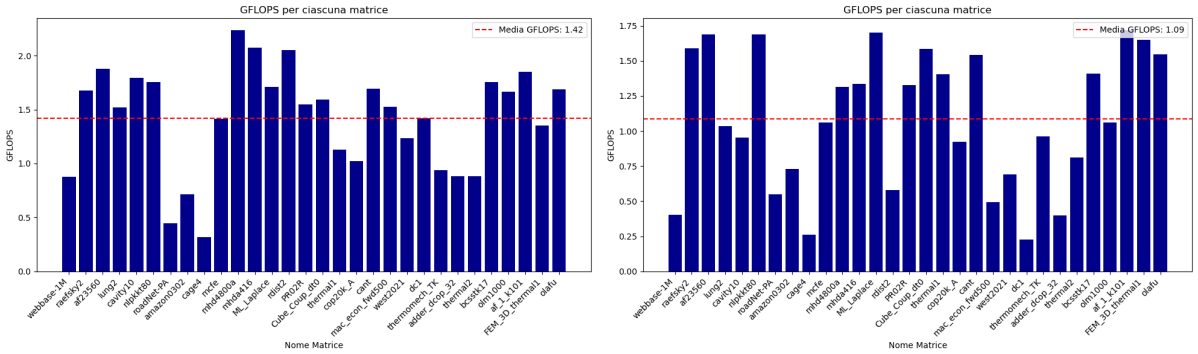
9 Risultati

Nella seguente sezione verranno analizzati le prestazioni ottenute nell'esecuzione dei vari nuclei di calcolo analizzati precedentemente il prodotto matrice sparsa-vettore implementati in OpenMP e in CUDA utilizzando i formati `CSR` e `HLL`.

9.1 Esecuzione Seriale

Nelle seguenti immagini vengono mostrate le prestazioni del calcolo seriale dell'operazione considerata utilizzando le matrici di collaudo sia nel formato `CSR` che nel formato `HLL`.
Come si può analizzare dai dati raccolti si nota che, in media, utilizzando il formato `CSR` si ottengono prestazioni superiori rispetto al formato `HLL`. La differenza è dovuta alla maggiore efficienza in termini di memoria del formato `CSR` dove vengono memorizzati solo i valori non nulli (in particolare `AS`, utilizzando gli indici di colonna `JA` e l'offset di riga `IRP`) senza dover considerare alcun tipo di overhead dovuto a zeri aggiuntivi.
Il formato `HLL`, nonostante l'utilizzo degli array monodimensionali, utilizza comunque del padding locale e non si ha un accesso sequenziale ai dati dato che sono organizzati in blocchi con la presenza di un overhead aggiuntivo.
In confronto, quindi, il formato `CSR` grazie alla sua maggiore compattezza e semplicità di ac-

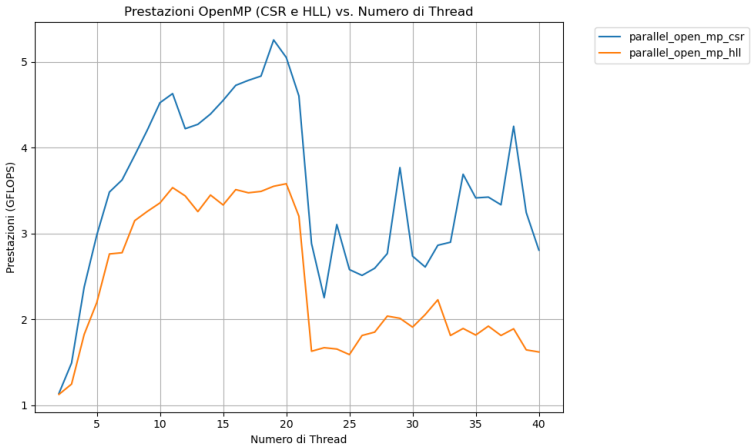
cesso ai dati risulta più adatto per sfruttare al meglio l'esecuzione seriale e la memoria cache, portando a prestazioni superiori.



Va indicato che per come sono costruiti i formati, il comportamento non è del tutto inaspettato o insolito, questo è prorio dovuto a come differiscono i due formati e quindi al tipo di accesso ai dati stessi. Con il formato `CSR` non si devono gestire aspetti di overhead che effettivamente vengono considerati con `HLL`.

9.2 OpenMP

Con OpenMP, sia utilizzando il formato `CSR` che il formato `HLL`, si è utilizzato un numero incrementale di thread utilizzati, in particolare da 2 a 40 thread per il calcolo del prodotto matrice sparsa-vettore.
Per entrami i formati utilizzati con i nuclei di calcolo precedentemente descritti non si hanno delle notevoli differenze in termini di prestazioni. Per entrambi i formati si ha una fase in cui si raggiungono notevoli prestazioni nel caso in cui si utilizzano dai 12 ai 20 thread, per poi avere un netto calo delle prestazioni.
In generale questo comportamento è dovuto al fatto che con l'aumentare del numero di threads utilizzato non si ottiene un aumento lineare delle prestazioni, comportamento dovuto al fatto che aumentano le operazioni di gestione: con l'aumentare del numero di thread si ha un overhead nella gestione da parte dell'hardware.
Nella prossima figura si possono notare le prestazioni medie ottenute con entrambi i formati al variare del numero di thread utilizzati.

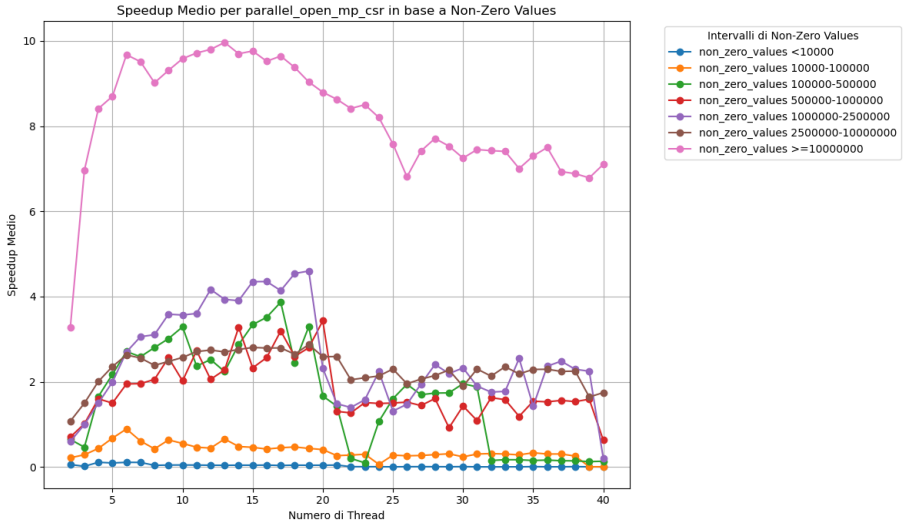


Questo comportamento può essere giustificato anche dall'architettura della CPU utilizzata per la raccolta delle prestazioni. L'Intel Xeon Silver 4210, infatti, dispone di 10 core fisici e supporta fino a 20 thread grazie all'Hyper-Threading, ovvero che il processore può gestire due thread per

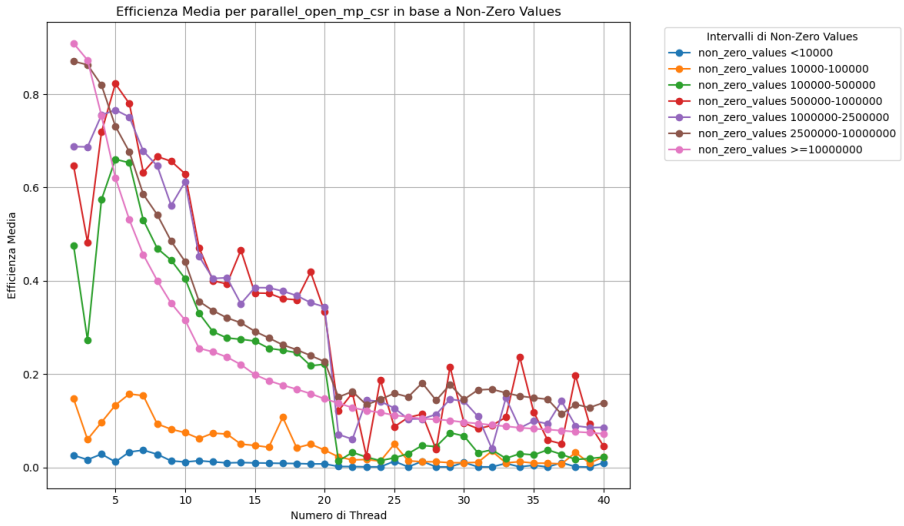
core, e quindi se inizialmente l'aggiunta di più thread può migliorare le prestazioni sfruttando il parallelismo, ma c'è anche un limite oltre il quale i benefici iniziano a diminuire con il conseguente decremento delle prestazioni.
Il fatto che il picco delle prestazioni si verifica tra 12 e 20 thread utilizzati suggerisce che i core fisici e i thread associati stiano lavorando al massimo delle loro capacità, sfruttando al meglio le risorse a disposizione della CPU.
Quando poi il numero di thread supera il numero di core fisici, in questo caso il bus di memoria che collega i componenti di memoria della CPU diventa il collo di bottiglia dato che l'utilizzo di più thread comporta alla saturazione del componente e al conseguente degrado delle prestazioni dovuto all'overhead dovuto al parallelismo.

9.2.1 CSR

Dopo aver analizzato l'andamento delle prestazioni, al variare del numero dei thread, a si è eseguito un ulteriore caso di studio: l'impatto della densità di elementi non nulli nella matrice. In particolare, è emerso come le matrici caratterizzate da una più alta concentrazione di nonzeri, come ad esempio quelle comprese tra 100.000 e 500.000, oppure superiori ai 10 milioni, evidenziano prestazioni decisamente migliori rispetto a quelle con una presenza più limitata di elementi non nulli.
Tale risultato può essere apprezzato nelle successive figure:



Questa prima figura illustra l'andamento dello speedup in funzione del numero di thread utilizzati e della quantità di zeri presenti nella matrice.
Analizzando il grafico, si osserva un comportamento piuttosto chiaro: nelle fasi iniziali, con un numero limitato di thread, lo speedup cresce rapidamente, in particolare per le matrici con una maggiore densità di elementi non nulli.
Questo indica che l'algoritmo riesce a sfruttare efficacemente la parallelizzazione in questi casi. Tuttavia, superata la soglia dei 20 thread, il trend si inverte bruscamente: lo speedup subisce un forte calo, per poi stabilizzarsi quando si supera il limite dei 27 thread.
Questo comportamento è indicativo di un crescente overhead derivante dalla gestione dei thread e dalle operazioni di sincronizzazione, il cui impatto diventa predominante al superamento di una determinata soglia di parallelismo.
Diversamente, per le matrici con un numero ridotto di elementi non nulli — ad esempio negli intervalli tra 10.000 e 50.000 o inferiori a 10.000 — le prestazioni rimangono piuttosto stabili, senza variazioni significative né in aumento né in diminuzione.



La figura soprastante, invece, ci mostra un altro aspetto da non sottovalutare: l'efficienza, che, come detto precedentemente, misura l'utilizzo effettivo delle risorse di calcolo rispetto a un'esecuzione ideale, in cui ogni processore contribuisce in modo perfettamente lineare. Dal grafico si osserva un comportamento abbastanza uniforme indipendentemente dalla densità di zeri nella matrice: in tutti i casi, l'efficienza tende a raggiungere un picco nelle fasi iniziali, quando il numero di thread è contenuto, per poi diminuire progressivamente al crescere del grado di parallelismo.
Le uniche differenze legate al numero di elementi non nulli nella matrice riguardano esclusivamente l'intensità del calo di efficienza osservato: per matrici con un elevato numero di elementi non nulli, la riduzione risulta particolarmente marcata, passando da un massimo di circa 0,92 fino a un minimo di 0,09, evidenziando un comportamento addirittura critico. Tale dinamica tende invece a stabilizzarsi con il diminuire del numero di non zeri, mostrando un andamento più regolare e meno drastico.
Questo calo è principalmente attribuibile all'overhead di gestione dei thread e alle operazioni di sincronizzazione, che diventano sempre più onerose man mano che il numero di thread supera una determinata soglia. In particolare, la gestione della concorrenza perde efficienza con l'aumentare dei thread, riducendo di conseguenza le prestazioni complessive.

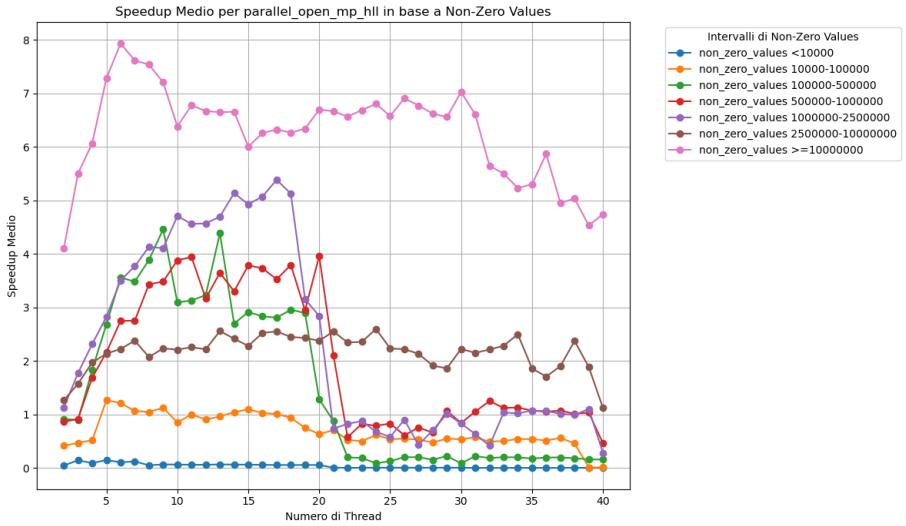
9.2.2 HLL

Uno degli aspetti fondamentali da considerare, oltre alle prestazioni ottenute, è l'impatto dei valori non nulli della matrice, aspetto considerato anche per il formato `CSR`.
Bisogna considerare che le prestazioni ottenute sono basate sull'implementazione basata sugli array monodimensionali indicata precedentemente.
Il formato `HLL` mostra dei comportamenti simili al formato `CSR`, nonostante siano differenti in termini di memorizzazione della matrice. In particolare, si nota come per matrici con un numero di valori non zeri superiore a 10'000'000 si ottengono prestazioni migliori rispetto alle altre tipologie di matrici.
Si può notare come che per quasi tutte le tipologie di matrici si ha un comportamento molto simile: si ha una fase in cui lo speedup aumenta per poi avere una fase di decadimento. La fase in cui si ha in incremento dello speedup corrisponde in quei casi di esecuzione in cui con l'aumentare del numero di thread in cui il nucleo di calcolo utilizzato riesce a sfruttare la capacità di parallelismo del processore.
I valori di speedup massimi, infatti, vengono raggiunti nel range tra i 12 e i 20 thread utilizzati, range che si era già notato quando si sono valutate le prestazioni medie dei nuclei di calcolo, ovvero quei range dove la CPU utilizzata riesce a gestire correttamente i thread senza avere dei

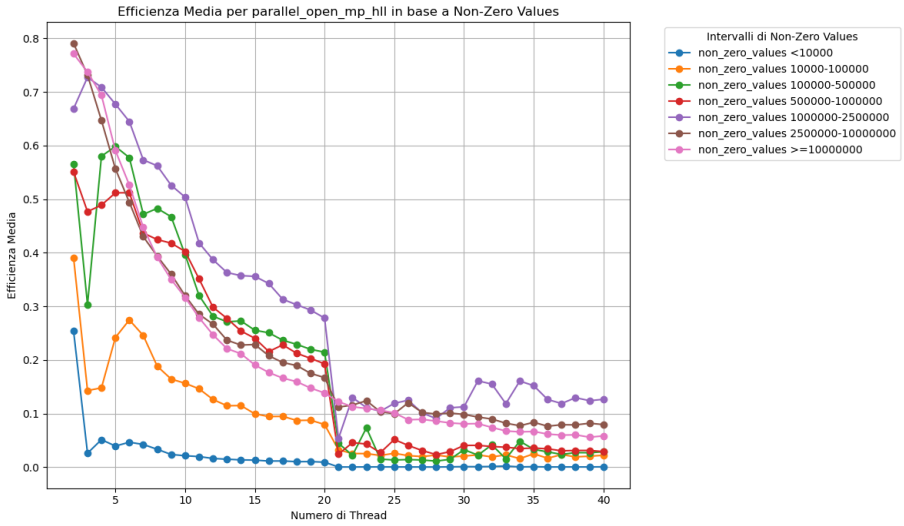
colli di bottiglia all'interno dei componenti.
Questo comportamento è abbastanza visibile per le seguenti categorie di matrici:

- matrici con un numero di non zeri comprso tra 100'000 e 500'000 (linea verde nel prossimo grafico);
- matrici con un numero di non zeri comprso tra 500'000 e 1'000'000 (linea rossa nel prossimo grafico);
- matrici con un numero di non zeri comprso tra 1'000'000 e 2'500'000 (linea viola del prossimo grafico);

Nella prossima figura viene raffigurato l'andamento dello speedup al variare del numero di thread utilizzati per il prodotto matrice sparsa-vettore.

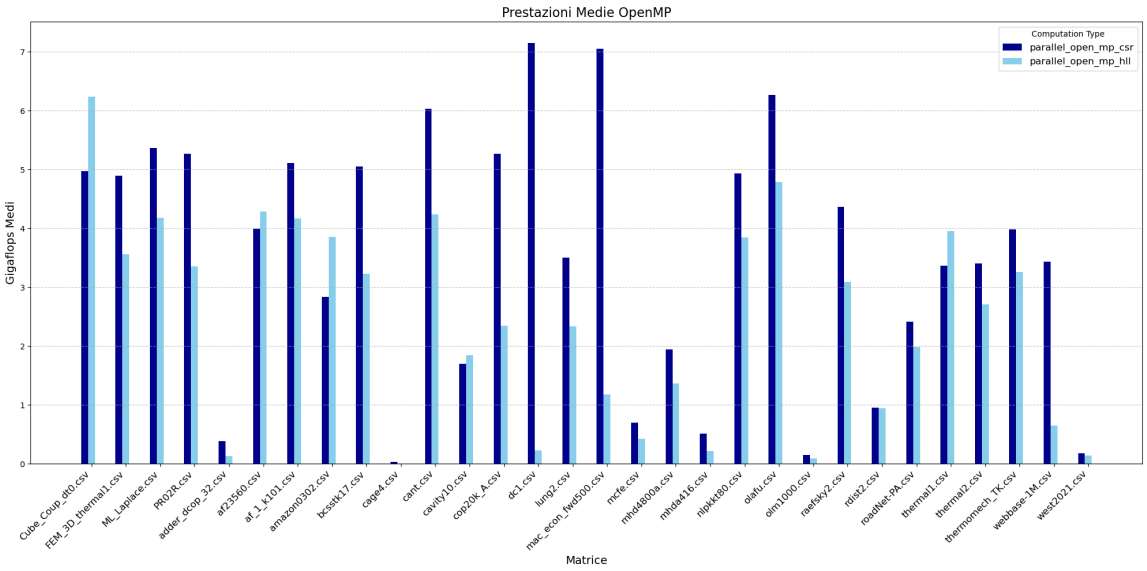


Per le matrici con un numero di valori non nulli maggiore di 10'000'000, invece, i valori di speedup più alti si hanno quando il numero di thread utilizzato è piuttosto basso, ovvero nell'intervallo tra i 4 e gli 8 thread. Successivamente si ha una prima fase di decadimento dello speedup per poi avere una fase dove si ha un andamento stabile. Superata la soglia dei 30 thread utilizzati si ha un decadimento importante del valore dello speedup. Questo perché con l'utilizzo di un numero troppo grande di thread si perdono i benefici introdotti dal parallelismo.
Per le matrici con un numero non troppo grande di valori non nulli, ovvero per quelle matrici con un numero di valori non nulli fino a 10'000, si ha un valore dello speedup abbastanza stabile.
Per le matrici con un numero di non zeri compreso tra 10'000 e 100'000 si ha una faso in cui lo speedup cresce per poi decrescere quando si utilizzano più di 20 thread.
Per le matrici con un numero di valori non nulli compreso nel range 2'500'000 - 10'000'000 si ha una prima fase in cui lo speedup cresce per poi assumere un valore abbastanza stabile per poi diminuire quando di perdono i benefici del parallelismo.
Un altro aspetto da tenere in considerazione è l'efficienza.
In generale si nota un comportamento abbastanza uniforme: per tutte le categorie di matrici il picco di efficienza si ha quando il numero di thread utilizzato è abbastanza basso, per poi avere una fase di decrescita costante fino al caso in cui si utilizzano 20 thread, per poi avere una netta diminuzione superata questa soglia.
L'andamento dell'efficienza nota come che con l'aumentare del numero di thread si perdano i benfici introdotti dal parallelismo.
Nella seguente figura viene mostrato il valore dell'efficienza al variare del numero di thread utilizzato in base al numero di valori non zeri della matrice.



9.2.3 Prestazioni

Nella seguente figura vengono rappresentate le prestazioni medie ottenute utilizzando le matrici di collaudo utilizzate considerando il calcolo sia con il formato `CSR` e il formato `HLL`.



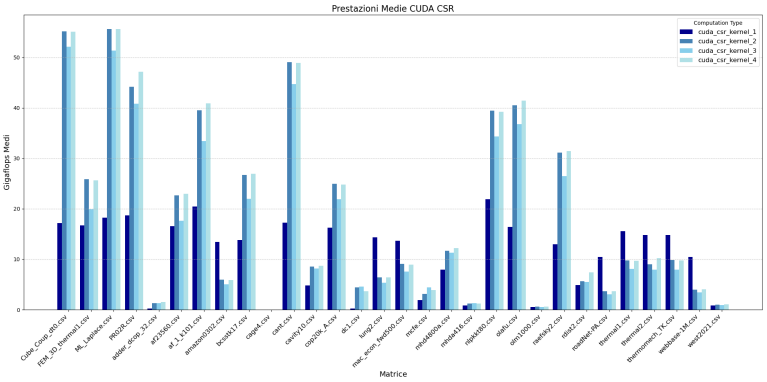
In generale si nota come che utilizzando OpenMP le prestazioni medie ottenute con il formato `CSR` siano di gran lunga migliori rispetto alle prestazioni medie ottenute rispetto all'utilizzo del formato `HLL`.
Una motivazione ai risultati ottenuti consiste nella differenza effettiva dei formati di memorizzazione, in particolare il formato `CSR` registra direttamente i valori diversi da zero considerando direttamente le posizioni all'interno degli array utilizzati per salvare i puntatori all'inizio di ogni riga e al vettore degli indici di colonna. Con `HLL` si paga l'aggiunta del padding e l'organizzazione in blocchi della matrice nonostante l'utilizzo degli array monodimensionali.

9.3 CUDA

In questa sezione verranno analizzate le prestazioni ottenute con CUDA, analizzando le prestazioni ottenute con i differenti kernel utilizzati per i formati `CSR` e `HLL`.

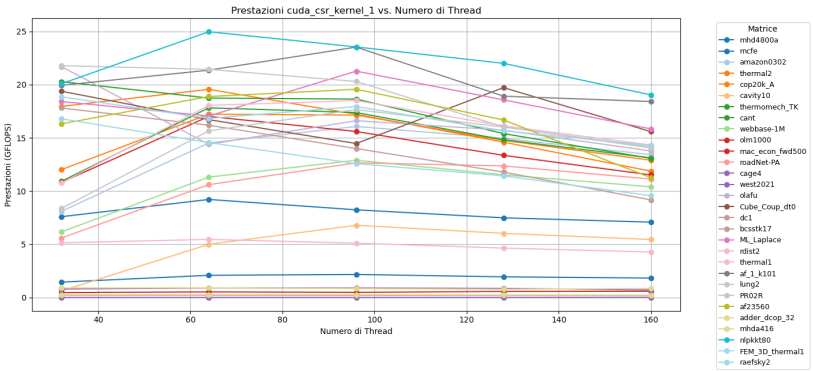
9.3.1 CSR

Tra le diverse tecniche di ottimizzazione per il prodotto matrice-vettore (SpMV) in formato `CSR` utilizzando CUDA, concentrando l'interesse sui quattro kernel illustrati precedentemente. Nel grafico sottostante, sono mostrati i valori medi di prestazioni in GigaFlops per ogni matrice testata, con riferimento al numero di kernel eseguiti all'interno di un singolo blocco di calcolo. Il confronto, nella figura sottostante, ci offre l'opportunità di analizzare l'efficacia di ciascuna implementazione, al variare delle matrici.

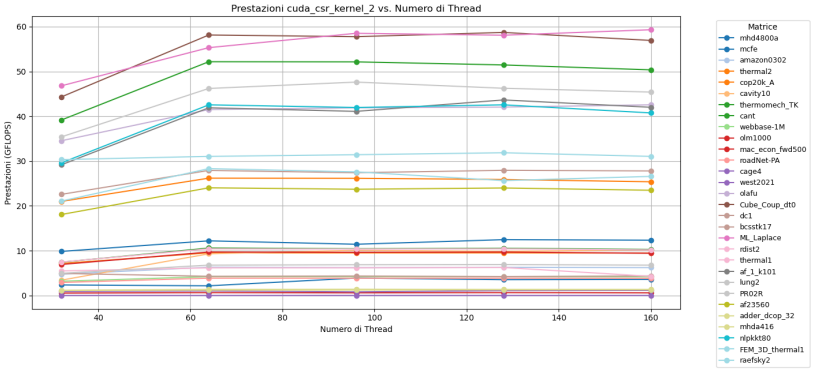


Ciò che può essere concluso in generale su tutti i kernel è che le loro migliori performance vengono sempre raggiunte con matrici caratterizzate da un elevato numero di elementi non nulli, come `Cube_Coup_dt0`, `ML_Laplace` e `af_1_k101`. I risultati ottenuti sono i seguenti:

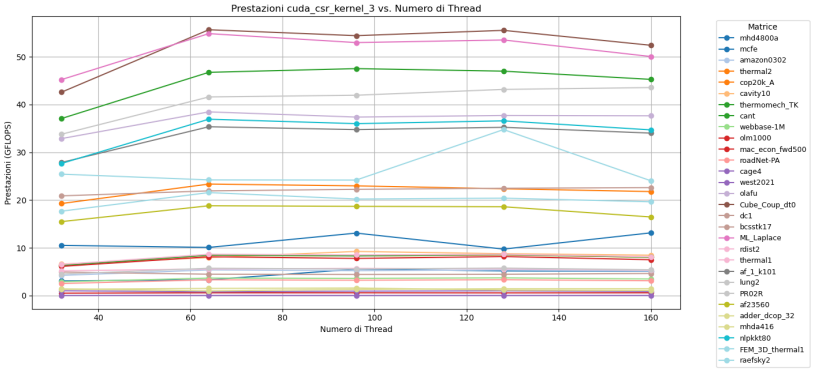
1. **csr_matvec_kernel**: questa soluzione rappresenta un approccio di base, caratterizzato dall'assegnazione di un thread per ogni riga e dall'accesso diretto alla memoria globale. Sebbene l'implementazione sia relativamente semplice, essa presenta limitazioni dovute al fatto che gli accessi non sono coalescenti, in particolare quando le righe contengono un numero ridotto di elementi. Le prestazioni di questo kernel si attestano su valori inferiori rispetto a soluzioni più evolute. In particolare, la configurazione con 64 thread si distingue per il miglior risultato in termini di GFlops massimi raggiunti, mentre, considerando le prestazioni medie su tutte le matrici, l'impiego di 96 thread si rivela più equilibrato, assicurando una maggiore efficienza nell'esecuzione;



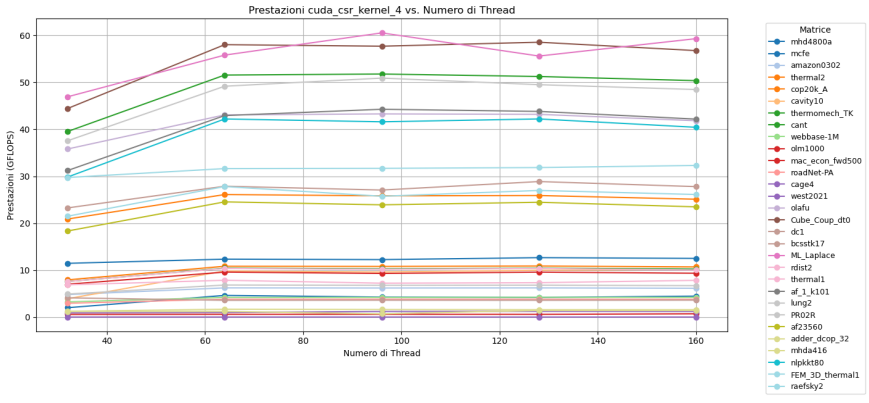
2. **csr_matvec_shfl_reduction**: questa soluzione è un approccio intermedio, assegnando l'elaborazione di ciascuna riga a un singolo gruppo di thread. Raggiunge le migliori prestazioni con matrici caratterizzate da un elevato numero di elementi non nulli e con un numero di warp pari a 3 (ognuno composto da 32 thread), per un totale di 96 thread;

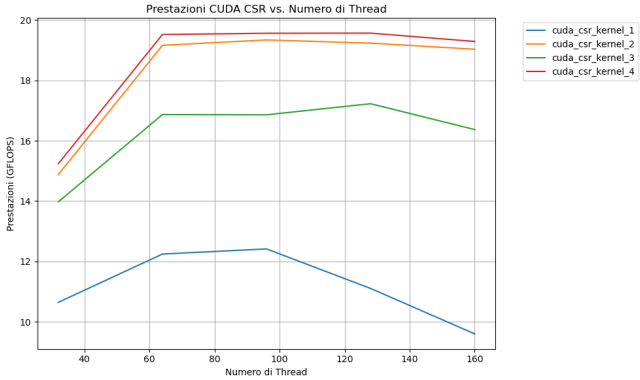


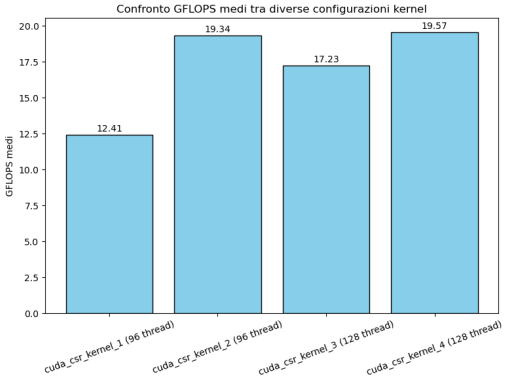
3. **csr_matvec_shared_memory**: il terzo kernel mantiene la struttura di quello precedente, ma ha la peculiarità di integrare l'uso della memoria condivisa. Questo permette coalescenza negli accessi, ma non la garantisce in pieno, accumulando i risultati parziali del prodotto e riducendo l'accesso alla memoria globale per il vettore risultato y. In questo caso, le prestazioni ottimali si ottengono con l'uso di 4 warp;



4. **csr_matvec_warp_cacheL2**: come nei secondi e terzi kernel, viene assegnato un warp per riga; tuttavia, in questo caso si sfrutta `__ldg()` per migliorare l'efficienza degli accessi alla cache L2, riducendo la latenza grazie a una migliore locality. Questa soluzione evita l'uso della memoria condivisa. In particolare, è il kernel che ottiene i migliori risultati, sia in termini medi che assoluti, e, utilizzando la configurazione con 3 warp, raggiunge il picco massimo di **60.541 GFlops** nella matrice `ML_Laplace`, al contempo le prestazioni in media su tutte le matrici, suggeriscono l'utilizzo di 128 thread;

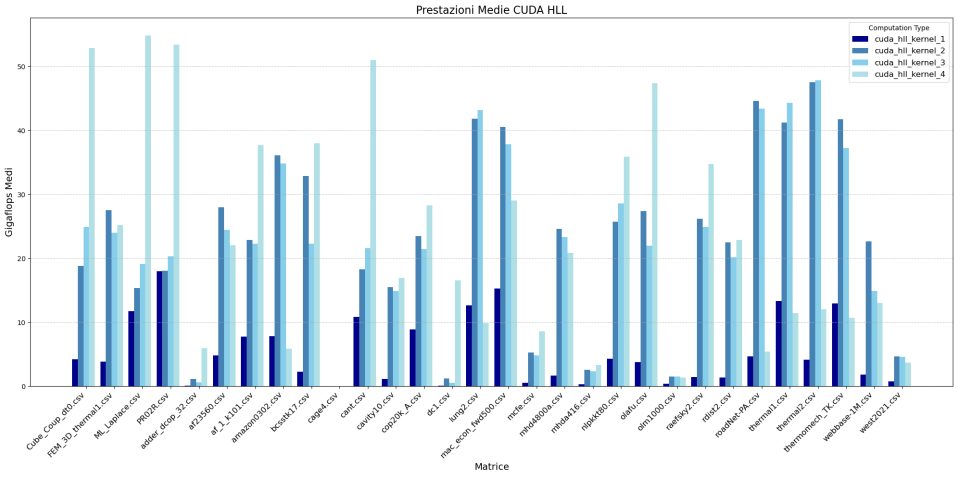






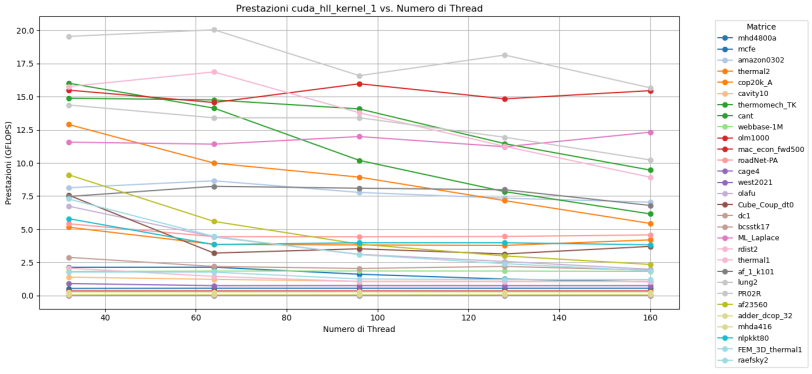
9.3.2 HLL

Nella seguente figura vengono mostrati con un confronto diretto i valori medi dei diversi kernel CUDA utilizzando il formato `HLL`, valori ottenuti con le differenti iterazioni in termini di warp utilizzati per il calcolo.



In generale di può notare che:

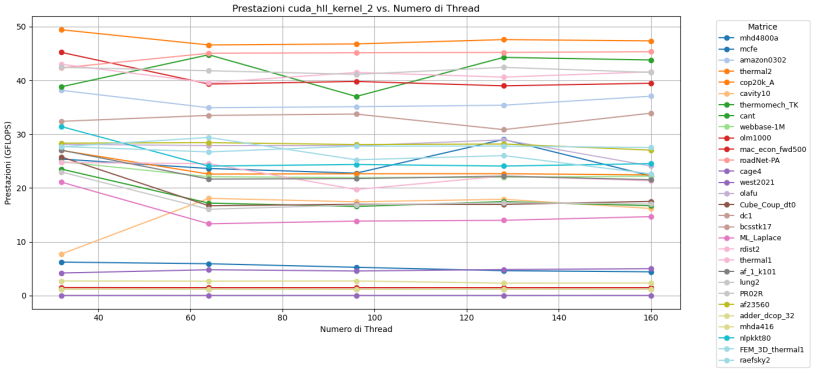
- il kernel `cuda_hll_kernel_v1` rappresenta una soluzione molto semplice, infatti si arriva al massimo ad un valore non superiore ai 20 GigaFlops medi. È un risultato aspettato data la semplicità del kernel, si utilizza infatti un thread per hack, considerando anche l'utilizzo della memoria globale, tipologia di memoria che porta maggiori latenze.



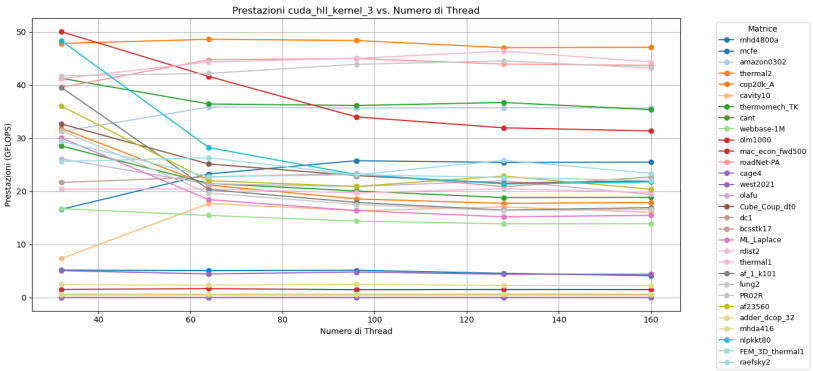
Nella figura precedente sono mostrate le ottenute sulle matrici di collaudo al variare del numero di thread utilizzato. Come si può notare anche dalle prestazioni non si ha un

comportamento uniforme, si nota però che con l'aumentare del numero di thread utilizzato le prestazioni calcano per la maggior parte di matrici;

- il kernel `cuda_hll_kernel_v2` rappresenta una miglioria del precendente kernel, infatti si utlizza un thread per riga. Anche in questo caso si utilizza una memoria globale.
  Nonostante il fatto che si utilizzi un tipo di memoria costosa in termini di latenze di accesso, si nota come che per alcune matrici di collaudo porta i risultati migliori in termini di prestazioni, arrivando a picchi di prestazioni medie di circa 48 GigaFlops.
  Considerando le prestazioni ottenute con le matrici di collaudo si nota come non si ha un comportamento uniforme.
  Le prestazioni, per alcune matrici, sono indipendenti dal numero di thread utilizzati; per altre con l'aumentare del numero di thread utilizzati si ha una fase in cui si ottengono dei miglioramenti per poi avere un valore costante in termini di prestazioni ottenute, mentre per altre matrici si ha il comportamento opposto, ovvero una prima fase in cui le prestazioni peggiorano per poi avere un valore costante con l'aumentare del numero di thread utilizzati.
  Nella prossima figura vengono rappresentate le prestazioni del secondo kernel testato sulle matrici di collaudo.

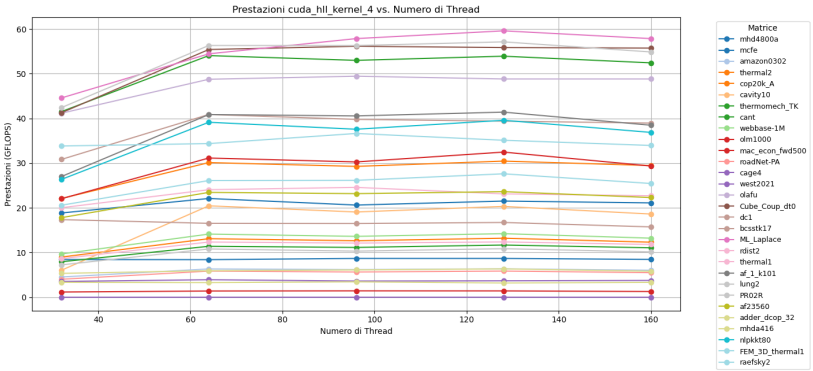


- il terzo kernel, `cuda_hll_kernel_v3`, utilizza un approccio basato sulla memoria condivisa per ridurre le latenze alla memoria globale. In un paio di casi rappresenta la soluzione in cui si ottengono le prestazioni medie migliori. Bisogna sottolineare che in questa implementazione non sono stati utilizzati degli approcci ottimizzati in termini di riduzione, ma solo in termini di accesso alla memoria utilizzando la memoria condivisa al posto della memoria globale.



Come per il kernel precedente, non si ha un comportamento uniforme per le matrici di collaudo utilizzate.

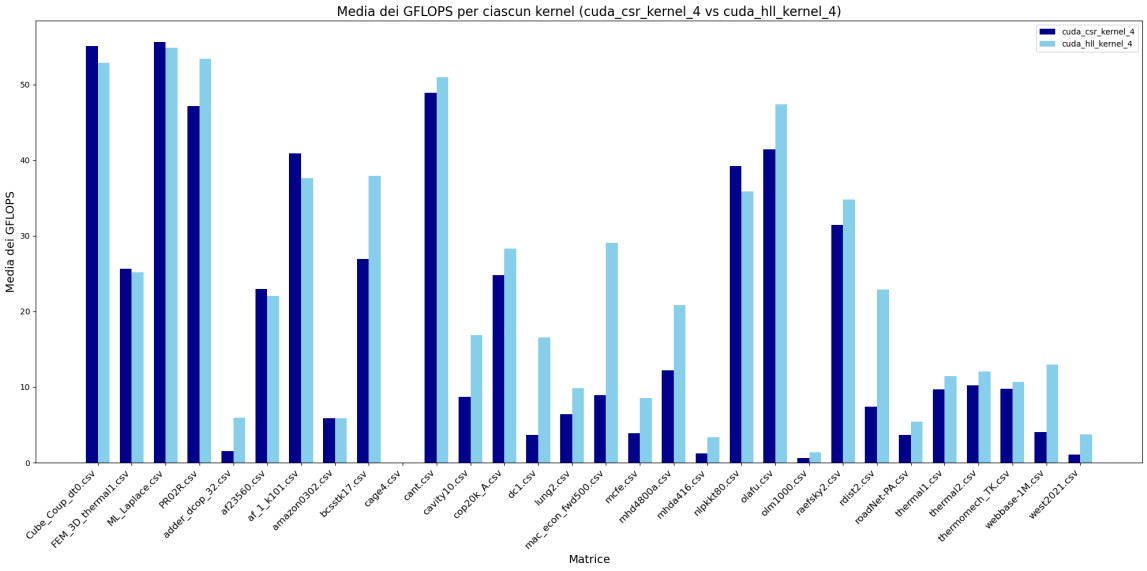
- il kernel `cuda_hll_kernel_v4` rappresenta il kernel migliore per la maggior parte delle matrici. In questo kernel viene utilizzato una configurazione basata su memoria globale utilizzando un warp per riga utilizzando la funzione `__shfl_down_sync()` come metodo di riduzione. Un altro motivo del perché si ottengono prestazioni medie così alte è grazie agli accessi coalescenti ai dati in memoria da perte dei thread.
  Un'altra ottimizzazione utilizzata consiste nell'utilizzo degli operatori bitwise del `C`, questi operatori possono portare a dei miglioramenti in termini di efficienza.
  Grazie a queste caratteristiche, rappresenta il kernel che per la maggior parte delle matrici consente di ottenere le prestazioni medie migliori, nonostante ci siano dei casi in cui non sia effettivamente il kernel che consente di ottenere le prestazioni medie migliori.
  Nel prossimo grafico si nota come per la maggior parte delle matrici di collaudo utilizzate si ha un miglioramento delle prestazioni con l'aumentare del numero di thread utilizzati, mentre in quei casi in cui non si ha un miglioramento non si hanno peggioramenti in termini di prestazioni raggiunti ma si mantiene un valore costante in termini di prestazioni ottenute.



In conclusione, si può decretare che il kernel migliore per il formato di memorizzazione `HLL` sia l'ultimo kernel analizzato dato che raggiunge i risultati migliori in termini di prestazioni raggiunte.

9.4 Best CSR vs Best HLL

I risultati riportati nei succesivi grafici evidenziano come il kernel basato su formato `CSR` raggiunga picchi prestazionali leggermente superiori rispetto al kernel `HLL`, per matrici caratterizzate da un elevato numero di elementi non nulli, come `Cube_Coup_dt0`, `ML_Laplace` e `af_1_k101`. Tuttavia, nell'analisi complessiva, il kernel che utilizza il formato `HLL` si dimostra globalmente più efficiente: pur registrando prestazioni di poco inferiori rispetto a `CSR` per matrici di grandi dimensioni, esso risulta nettamente superiore su matrici di scala più contenuta. Fra queste ultime, spiccano: `mac_econ_fwd500`, `rdist2`, `dc1e`, e `mhda416`.
Mediamente il formato `HLL` mostra un vantaggio più evidente per la maggior parte delle matrici di collaudo utilizzate suggerendo che il suo schema di gestione dei dati risulti essere più efficiente quando si utilizza CUDA, comportamento che nel caso dell'esecuzione parallela con OpenMP non si verificava, infatti si ottenevano i risultati opposti.
Un fattore chiave che influisce sulle prestazioni è proprio la distribuzione dei valori non nulli all'interno della matrice.
Una possibile giustificazione di questo comportamento può essere attribuita al fatto che utilizzando `HLL` si suddivide la matrice in blocchi in modo da ottimizzare il carico di lavoro e migliorando l'accesso alla memoria rispetto al formato di memorizzazione `CSR`.
Nel seguente grafico vengono rappresentate le prestazioni medie ottenute rispettivamente con il kernel 4 sia per `CSR` che per `HLL` sulle matrici di collaudo utilizzate.



Bisogna sottolineare che i risultati ottenuti non hanno saturato completamente la potenza a disposizione della GPU.
Questo approccio è stato utilizzato a seguito dei primi risultati sperimentali raccolti. Dopo aver controllato ogni volta che il vettore risultante calcolato sia corretto valutandone l'errore, si è verificato che saturando la potenza computazionale a disposizione non si è ottenuto un beneficio in termini di prestazioni. on l'aumentare del numero di thread si notava un incremento dell'errore del vettore calcolato. In particolare si è notato come con i kernel implementanti, e le configurazioni utilizzate, si raggiungono dei buoni compromessi in termini di efficienza e correttezza nell'esecuzione dell'operazione.

10 Suddivisione del Lavoro

- Preprocessamento dati, conversione in formato CSR ed HLL: Alessandro Cortese
- Prodotti seriali nel formato CSR ed HLL: Luca Martorelli ed Alessandro Cortese
- OpenMP with CSR: Luca Martorelli
- OpenMP with HLL: Alessandro Cortese
- CUDA with CSR: Luca Martorelli
- CUDA with HLL: Alessandro Cortese
- Array List format to CSV: Alessandro Cortese
- Graph plotting: Luca Martorelli
- Reporting: Luca Martorelli ed Alessandro Cortese