



# UNIVERSITÀ DEGLI STUDI DI ROMA TOR VERGATA

MACROAREA DI INGEGNERIA

Corso di Parallel Computing Systems and Applications

## Relazione di progetto

Alexandru Nazare & Luca Cuppellaro

[https://github.com/NomeMio/scpa\\_Ax](https://github.com/NomeMio/scpa_Ax)

2024-2025

# Indice

<b>1</b>	<b>Introduzione</b>	<b>3</b>
1.1	Descrizione del problema . . . . .	3
1.1.1	Formati di Rappresentazione delle Matrici . . . . .	3
1.1.2	Approcci Algoritmici . . . . .	4
1.1.3	Complessità e Strumenti della Parallelizzazione . . . . .	4
1.1.4	Obiettivo Finale . . . . .	4
1.2	Struttura del progetto e come utilizzarlo . . . . .	5
1.2.1	Compilazione ed Esecuzione Principale . . . . .	6
1.3	Strumenti utilizzati . . . . .	7
<b>2</b>	<b>Soluzioni sviluppate</b>	<b>8</b>
2.1	Caricamento dati e dataset . . . . .	8
2.1.1	Dataset di Matrici Utilizzato . . . . .	8
2.1.2	Lettura e Rappresentazione dei Dati . . . . .	9
2.2	Implementazione del Formato HLL per Matrici Sparse . . . . .	9
2.2.1	Descrizione del Formato HLL . . . . .	9
2.2.2	Calcolo Seriale . . . . .	11
2.2.3	Calcolo Parallelo con OpenMP . . . . .	11
2.2.4	Calcolo Parallelo con CUDA . . . . .	13

2.3	Implementazione del Formato CSR per Matrici Sparse . . . . .	18
2.3.1	Descrizione breve del formato CSR . . . . .	18
2.3.2	Calcolo Seriale . . . . .	18
2.3.3	Calcolo Parallelo con OpenMP . . . . .	19
2.3.4	Calcolo Parallelo con CUDA . . . . .	20
2.3.4.1	Gestione del Warp: Shuffle e Riduzione Diretta . . .	23
<b>3</b>	<b>Performance delle soluzioni</b>	<b>27</b>
3.1	Spiegazione delle misurazioni . . . . .	27
3.2	Architettura su cui abbiamo testato . . . . .	27
3.3	Analisi sui dati CPU . . . . .	28
3.4	Analisi sui dati GPU . . . . .	30
3.5	Comparazione dei risultati . . . . .	34
3.6	Dati tabellari . . . . .	40
<b>4</b>	<b>Conclusioni</b>	<b>42</b>
4.1	Suddivisione lavoro . . . . .	43

# Capitolo 1

## Introduzione

### 1.1 Descrizione del problema

Il problema fondamentale affrontato in questo progetto è il calcolo del prodotto tra una matrice  $A$  e un vettore  $x$ :

$$y \leftarrow Ax$$

Si tratta di un'operazione che chiunque può, in linea di principio, affrontare con carta e penna (oppure eseguendo i calcoli a mente). Tuttavia, questo approccio manuale presenta evidenti limiti di *scalabilità*: all'aumentare delle dimensioni della matrice e del vettore, il calcolo diventa rapidamente intrattabile per un essere umano. Data la notevole importanza di questa operazione in numerosi contesti scientifici e ingegneristici, emerge la necessità di automatizzarne l'esecuzione.

Per tale automazione, ci affidiamo alle macchine (calcolatori) che eseguono algoritmi da noi definiti. È però cruciale prestare attenzione sia alla *rappresentazione dei dati* (come le matrici vengono memorizzate) sia agli *algoritmi* stessi, al fine di utilizzare in modo efficiente le *risorse di computazione* disponibili.

#### 1.1.1 Formati di Rappresentazione delle Matrici

Per quanto riguarda la rappresentazione dei dati, in questo progetto le matrici verranno memorizzate utilizzando due formati specifici per matrici sparse (matrici con molti elementi nulli):

- **Compressed Sparse Row (CSR)**: Un formato largamente utilizzato che memorizza solo gli elementi non nulli della matrice, insieme agli indici di colonna corrispondenti e a puntatori all'inizio di ogni riga.
- **Hybrid with Localized Lists (HLL)**: Il formato Hybrid with Localized Lists (HLL) combina l'efficienza del formato ELLPACK con una suddivisione in blocchi ("hacks") per ridurre il padding in matrici sparse eterogenee. La matrice è divisa in gruppi di righe (hackSize) chiamati blocchi ELLPACK, ciascuno gestito come sottomatrice in formato ELLPACK. Ogni blocco memorizza elementi non nulli e indici di colonna in array linearizzati.

### 1.1.2 Approcci Algoritmici

Per quanto riguarda gli algoritmi utilizzati per effettuare il calcolo  $y \leftarrow Ax$ , abbiamo implementato e analizzato due approcci fondamentali. Sebbene l'idea di base del calcolo rimanga la stessa per entrambi i formati di matrice (CSR e HLL), l'implementazione pratica differisce e può portare a prestazioni notevolmente diverse a seconda del formato e dell'approccio scelti:

- **Seriele:** Rappresenta il modo sequenziale con cui normalmente si affronta un problema computazionale. Il problema viene suddiviso in *passi* (steps) di calcolo che vengono eseguiti uno dopo l'altro da una singola unità di elaborazione (ad esempio, un core della CPU). Questa entità opera sui dati senza doversi preoccupare, in generale, di problematiche legate alla concorrenza o alla coordinazione con altre unità.
- **Parallelo:** Costituisce l'approccio con cui, *idealmente*, si vorrebbe risolvere ogni problema computazionalmente intensivo. Anche in questo caso, il problema viene suddiviso in *passi*, ma questi vengono eseguiti, per quanto possibile, contemporaneamente (in parallelo) da più unità di elaborazione. Ciò può ridurre significativamente i tempi di esecuzione, ma introduce una maggiore complessità logica nell'algoritmo.

### 1.1.3 Complessità e Strumenti della Parallelizzazione

Sebbene la parallelizzazione di un problema appaia vantaggiosa, essa comporta diverse complicazioni aggiuntive: la necessità di *coordinazione* tra le diverse unità di elaborazione, la gestione degli *accessi concorrenti* a dati condivisi (per evitare race condition e garantire la coerenza) e, soprattutto, la scrittura di un algoritmo che sia non solo parallelizzabile, ma anche efficientemente mappato sulla piattaforma hardware/software scelta.

Per affrontare quest'ultima sfida nell'ambito di questo progetto, sono stati utilizzati due strumenti distinti per implementare gli algoritmi paralleli:

- **OpenMP:** Uno standard API (Application Programming Interface) che permette di creare applicazioni parallele C/C++/Fortran su sistemi a *memoria condivisa* (come le moderne CPU multi-core) utilizzando un insieme di direttive per il compilatore, funzioni di libreria e variabili d'ambiente.
- **CUDA-C:** Un'estensione del linguaggio C sviluppata da NVIDIA che permette di scrivere programmi eseguibili sulle sue *GPU* (*Graphics Processing Units*). Sfrutta l'architettura parallela massiva delle GPU per accelerare calcoli complessi.

### 1.1.4 Obiettivo Finale

Definiti i formati dei dati (CSR, HLL), gli approcci algoritmici (seriale, parallelo) e gli strumenti di parallelizzazione (OpenMP, CUDA), l'obiettivo finale di questo lavoro è

*misurare e confrontare* le performance (ad esempio, in termini di tempo di esecuzione o throughput) ottenute dall'esecuzione delle varie combinazioni possibili (es. CSR-Seriale, CSR-OpenMP, CSR-CUDA, HLL-Seriale, HLL-OpenMP, HLL-CUDA), al fine di valutare l'efficacia dei diversi formati e delle tecniche di parallelizzazione per il problema specifico del prodotto matrice-vettore.

## 1.2 Struttura del progetto e come utilizzarlo

Questa sezione illustra come è organizzato il codice sorgente del progetto all'interno delle diverse directory e fornisce indicazioni su come compilare ed eseguire i test principali, in particolare quelli dedicati alla misurazione delle performance.

La struttura delle directory principali è la seguente:

**lib/mat/** Questa directory rappresenta il cuore del progetto e contiene il codice sorgente della libreria sviluppata. Include:

- File header (.h) che definiscono le interfacce delle strutture dati (es. per i formati CSR e HLL) e delle funzioni implementate (es. `cuda_alex.h`, `cuda_luca.h`, `matrici0pp.h`, `mmio.h`, `stats.h`).
- I corrispondenti file sorgente (.c e/o .cu) con le implementazioni effettive delle funzioni per le operazioni sulle matrici, la gestione dei formati, la lettura/scrittura (`mmio`), il calcolo parallelo (CUDA, OpenMP) e la raccolta delle statistiche.
- È inoltre presente il file `CMakeLists.txt`, che definisce la configurazione per il sistema di build CMake, utilizzato per la gestione delle dipendenze e la compilazione del progetto.

**mat/** Questa directory contiene l'insieme delle matrici utilizzate per i test del progetto. Le matrici sono memorizzate nel formato standard Matrix Market (.mtx), i file possono anche essere dei link simbolici invece di file effettivi, questo per facilitare l'esecuzione su ambienti di calcolo condiviso fra più utenti.

**tests/** Questa directory raggruppa il codice sorgente e/o gli script utilizzati per testare le varie funzionalità implementate nella libreria `lib/mat/`.

- Il componente più rilevante all'interno di questa directory è `saveStats`. Si tratta dell'eseguibile principale dedicato al benchmarking. La sua funzione è quella di testare sistematicamente *tutte* le combinazioni significative di:
  - Algoritmi implementati (Seriale, OpenMP, CUDA).
  - Formati di matrice supportati (CSR, HLL).
  - Matrici di input presenti nella directory `mat/`.

Al termine dell'esecuzione, `saveStats` salva i risultati delle misurazioni per l'analisi successiva.

- Sono presenti altri test per testare le singole funzionalità ma sono di poca importanza.

**result/** In questa directory vengono raccolti i file generati contenenti i risultati grezzi delle misurazioni di performance, prodotti principalmente dall'esecuzione di **saveStats**. Oltre ai dati grezzi in formato CSV, questa cartella contiene anche script utilizzati per l'*elaborazione*, l'*analisi statistica* e la *visualizzazione grafica* dei risultati ottenuti.

/ Nella home del progetto sono presenti due file principali, il **Makefile** che permette di compilare ed eseguire il progetto e il file **AddMatrici.sh** che permette di importare in modo automatico le matrici del server di dipartimento dalla cartella **/data/matrici/**.

### 1.2.1 Compilazione ed Esecuzione Principale

Per la compilazione e la gestione delle dipendenze ci siamo affidati a CMake che, insieme a **make**, utilizziamo per compilare ed eseguire il nostro programma.

**Compilazione:** Per compilare il programma principale, che permette di eseguire tutte le componenti sviluppate, è sufficiente posizionarsi nella directory principale del progetto (dove si trova il Makefile principale) ed eseguire il comando:

```
make build-test-stats
```

Questo comando creerà una directory **build/** contenente l'eseguibile pronto per essere utilizzato.

**Preparazione delle Matrici:** Prima di eseguire i test, è necessario che le matrici desiderate siano presenti nella directory **mat/**. Questo può essere realizzato copiando i file delle matrici direttamente in questa cartella, oppure creando dei link simbolici che puntano alla loro posizione originale. Se si opera sul server del dipartimento, è disponibile uno script ausiliario **AddMatrici.sh**. Modificando l'array definito all'interno di questo script, è possibile specificare facilmente quali matrici includere o escludere dai test, eseguire questo script crea link simbolici a tutte le matrici specificate prendendole dalla cartella **/data/matrici/** del server di dipartimento.

**Esecuzione dei Test:** L'esecuzione dei test avviene tramite il target **make run-test-stats**, passando gli argomenti necessari tramite la variabile **ARGS**. Esistono due modalità principali:

- **Esecuzione su una matrice specifica:** Per eseguire i test solo su una particolare matrice (identificata da **cant.mtx** nell'esempio), usare un comando simile a:

```
make run-test-stats ARGS='2,4,8' 5 cant.mtx'
```

Gli argomenti specifici (qui "2,4,8" e 5) e l'identificativo della matrice (**cant.mtx**) vengono passati all'eseguibile.

- **Esecuzione su tutte le matrici selezionate:** Per eseguire i test su tutte le matrici definite nell'array dello script `AddMatrici.sh`, usare un comando simile a:

```
make run-test-stats ARGS='2,4,8' 5'
```

In questo caso, l'eseguibile ciclerà automaticamente su tutte le matrici configurate nello script, applicando gli argomenti forniti (qui "2,4,8" e 5).

Il primo argomento tra "" indica con quali numero di threads bisogna eseguire la parte OpenMP, per esempio "2,4,8" indica che andrà a testare prima con 2 poi con 4 e infine con 8. Il secondo argomento invece indica quante iterazioni dovrà effettuare l'applicazione per ogni configurazione. In ogni caso l'esecuzione produrrà un file `csv` nella cartella `results`

### 1.3 Strumenti utilizzati

Per lo sviluppo, la compilazione, l'analisi e la gestione del progetto sono stati impiegati i seguenti strumenti:

- **Build System:** CMake per la configurazione del progetto e la gestione delle dipendenze, e `make` per l'avvio del processo di compilazione e l'esecuzione di target specifici.
- **Linguaggi e Parallelismo:** C/C++ con estensioni **OpenMP** per il parallelismo su CPU e **CUDA-C** per la programmazione e l'ottimizzazione su GPU NVIDIA.
- **Controllo di Versione:** **Git**, ospitato sulla piattaforma **GitHub**, per il tracciamento delle modifiche e la collaborazione. Il repository del progetto è accessibile [qui](#).
- **Profiling CPU e Memoria:** `valgrind` è stato utilizzato per analizzare l'allocazione dinamica della memoria (tramite `malloc` e funzioni correlate) e per assicurare la corretta deallocazione di tutte le strutture dati, prevenendo memory leak.
- **Profiling GPU:** **NSight Compute** (fornito con il NVIDIA CUDA Toolkit nelle vecchie versioni o scaricabile a parte) è stato impiegato per il profiling dettagliato e l'analisi delle performance dei kernel CUDA eseguiti sulla GPU.
- **Elaborazione Dati e Visualizzazione:** **Python**, con l'ausilio di librerie come **Seaborn**, è stato utilizzato per processare i risultati dei benchmark, calcolare metriche statistiche (medie, deviazioni standard, speedup) e generare i grafici e le tabelle presentati nell'analisi delle performance.

# Capitolo 2

## Soluzioni sviluppate

### 2.1 Caricamento dati e dataset

#### 2.1.1 Dataset di Matrici Utilizzato

Le matrici che abbiamo utilizzato durante i test sono le seguenti:

- cage4.mtx
- mhda416.mtx
- mcfe.mtx
- olm1000.mtx
- adder\_dcop\_32.mtx
- west2021.mtx
- cavity10.mtx
- rdist2.mtx
- cant.mtx
- olafu.mtx
- Cube\_Coup\_dt0.mtx
- ML\_Laplace.mtx
- bcsstk17.mtx
- mac\_econ\_fwd500.mtx
- mhd4800a.mtx
- cop20k\_A.mtx
- raefsky2.mtx
- af23560.mtx
- lung2.mtx
- PR02R.mtx
- FEM\_3D\_thermal1.mtx
- thermal1.mtx
- thermal2.mtx
- thermomech\_TK.mtx
- nlpkkt80.mtx
- webbase-1M.mtx
- dc1.mtx
- amazon0302.mtx
- af\_1\_k101.mtx
- roadNet-PA.mtx

Questo insieme comprende matrici prevalentemente sparse, alcune delle quali simmetriche e altre con pattern specifici derivanti da diverse applicazioni scientifiche.

## 2.1.2 Lettura e Rappresentazione dei Dati

Per caricare i dati delle matrici in memoria, è stata implementata una funzione `loadMatRaw`. Questa funzione è responsabile della lettura dei file in formato Matrix Market (`.mtx`), avvalendosi della libreria C MMIO (Matrix Market I/O Library) per il parsing del formato.

La funzione carica i dati letti in una struttura intermedia denominata `MatriceRaw`, che memorizza la matrice nella seguente struttura:

```
typedef struct MatriceRaw {  
    unsigned int width, height, nz;                                ▷ Dimensioni e numero non-zeri  
    unsigned int *iVettore;                                         ▷ Vettore indici di riga  
    unsigned int *jVettore;                                         ▷ Vettore indici di colonna )  
    double *valori;                                              ▷ Vettore valori non-zeri  
} MatriceRaw;
```

Questa struttura `MatriceRaw` serve poi come punto di partenza per la conversione nei formati HLL e CSR, per la conversione sono state create delle funzioni apposite che si trovano nel file `matriciOpp.h`, che sono rispettivamente `convertRawToCsr`, `convertRawToHll`

## 2.2 Implementazione del Formato HLL per Matrici Sparse

In questa sezione viene presentata l'implementazione di un formato di memorizzazione efficiente per matrici sparse denominato HLL (Hybrid with Localized Lists). Questo formato è stato progettato per ottimizzare l'archiviazione e la manipolazione di matrici caratterizzate da un elevato numero di elementi nulli, sfruttando una struttura a blocchi di tipo ELLPACK (Ellpack-Itpack) combinata con informazioni sul numero di elementi non nulli (Nz). L'obiettivo principale di questa implementazione è ridurre l'ingombro di memoria e potenzialmente migliorare le prestazioni nelle operazioni che coinvolgono la matrice.

### 2.2.1 Descrizione del Formato HLL

Il formato HLL adottato in questo progetto si basa sulla suddivisione della matrice sparsa in blocchi orizzontali di dimensione fissa o variabile, determinata dinamicamente dal parametro `HackSize`. Ogni blocco viene memorizzato utilizzando una variante del formato ELLPACK, arricchita con metadati relativi al numero di elementi non nulli all'interno del blocco stesso. La struttura dati `MatriceHLL` definisce la rappresentazione completa della matrice in formato HLL:

Tipo Struttura `MatriceHLL`

```
totalRows: Intero  
totalCols: Intero  
HackSize: Intero  
numBlocks: Intero  
blocks: Array di puntatori a ELLPACK_Block
```

Come si può osservare, la struttura `MatriceHLL` contiene informazioni sulle dimensioni globali della matrice (`totalRows`, `totalCols`), la dimensione dei blocchi (`HackSize`), il numero totale di blocchi (`numBlocks`) e un array dinamico di puntatori a strutture `ELLPACK_Block`.

All'interno di ogni blocco ELLPACK:

```
Tipo Struttura ELLPACK_Block
M: Intero
N: Intero
MAXNZ: Intero
JA: Array di Interi
AS: Array di Reali
```

## Gestione del Numero di Elementi Non Nulli (Nz)

Una caratteristica distintiva del formato HLL implementato è la sua capacità di "salvare i valori Nz della matrice". Sebbene la struttura `ELLPACK_Block` come definita non contenga esplicitamente un campo per il numero di elementi non nulli per ogni riga, l'efficienza del formato HLL deriva dalla scelta di `MAXNZ` per dimensionare gli array `JA` e `AS`.

Il valore di `MAXNZ` per ciascun blocco è determinato dal numero massimo di elementi non nulli presenti in una qualsiasi delle sue righe. Questo approccio garantisce che tutti gli elementi non nulli del blocco possano essere memorizzati, evitando allocazioni dinamiche più complesse all'interno del blocco stesso. Tuttavia, è importante notare che questo può portare a uno spreco di memoria se le righe all'interno di un blocco hanno un numero di elementi non nulli significativamente variabile.

La scelta dinamica di `HackSize` è un aspetto chiave per bilanciare l'efficienza della memorizzazione e le prestazioni. Un `HackSize` troppo piccolo potrebbe aumentare il numero di blocchi e la complessità della gestione, mentre un `HackSize` troppo grande potrebbe portare a un `MAXNZ` elevato e a un maggiore spreco di memoria all'interno dei blocchi.

## Come viene calcolato l'hack size

Nel presente lavoro l'hack size, ovvero il numero di righe per blocco nel formato HLL/ELLPACK, viene selezionato in funzione del numero totale di elementi non nulli *nz* della matrice:

$$\text{hack} = \begin{cases} 16 & \text{se } nz < 10^4 \\ 32 & \text{se } 10^4 \leq nz < 10^5 \\ 64 & \text{se } nz \geq 10^5 \end{cases}$$

Questa scelta consente di adattare il parallelismo vettoriale ai registri SIMD disponibili (es. AVX2 e AVX-512), migliorando l'allineamento in memoria, saturando le pipeline SIMD e riducendo l'overhead di loop remainder. L'utilizzo di potenze di due facilita inoltre il caricamento efficiente nei registri vettoriali e il prefetch hardware.

## 2.2.2 Calcolo Seriale

Nel file `mat/hll.c` è implementato il calcolo seriale del prodotto matrice-vettore  $y \leftarrow Ax$ . La funzione prende in input:

- una matrice  $A$  memorizzata nel formato HLL,
- un vettore  $x$  di dimensione compatibile con  $A$ ,
- un vettore  $y$  inizializzato a tutti zeri.

La funzione calcola il prodotto matrice-vettore aggiornando il contenuto di  $y$ .

```
1: function SERIALMULTIPLYHLL(mat, vec, result)
2:   for b  $\leftarrow 0$  to mat  $\rightarrow$  numBlocks – 1 do
3:     blk  $\leftarrow$  mat  $\rightarrow$  blocks[b]
4:     start  $\leftarrow$  b  $\times$  mat  $\rightarrow$  HackSize
5:     maxNZ  $\leftarrow$  blk  $\rightarrow$  MAXNZ
6:     for i  $\leftarrow 0$  to blk  $\rightarrow M$  – 1 do
7:       t  $\leftarrow 0.0
8:       riga_base  $\leftarrow i \times maxNZ$ 
9:       for j  $\leftarrow 0$  to maxNZ – 1 do
10:        t  $\leftarrow t + AS[riga\_base + j] \times vec[JA[riga\_base + j]]$ 
11:       end for
12:       result[start + i]  $\leftarrow t$ 
13:     end for
14:   end for
15: end function$ 
```

## 2.2.3 Calcolo Parallelo con OpenMP

Nel file `mat/hll.c` è implementato il calcolo parallelo del prodotto matrice-vettore  $y \leftarrow Ax$  con la libreria OpenMP. La funzione `openMpMultiplyHLL` implementa il calcolo parallelo del prodotto matrice-vettore utilizzando la libreria OpenMP. La parallelizzazione viene effettuata su due livelli distinti: quello dei blocchi della matrice e quello delle operazioni di moltiplicazione e somma all'interno di ciascuna riga di un blocco.

- **Parallelizzazione a livello di righe:** Il ciclo interno che scorre sulle righe di ciascun blocco della matrice HLL è parallelizzato tramite la direttiva OpenMP pragma `omp parallel for schedule(static)`. Questa direttiva suddivide le righe del blocco tra i thread disponibili in modo statico: ogni thread elabora un sottoinsieme di righe. La scelta dello scheduling statico è particolarmente adatta quando il numero di operazioni per riga è simile, poiché garantisce una distribuzione equilibrata del carico di lavoro e un basso overhead di sincronizzazione.
- **Parallelizzazione SIMD delle operazioni interne al blocco:** All'interno del ciclo che processa le righe di un blocco, il ciclo interno che calcola il prodotto scalare per una singola riga (somma dei prodotti  $A_{ik}x_k$ ) viene ottimizzato con la direttiva `#pragma omp simd`. Questa direttiva suggerisce al compilatore di utilizzare istruzioni SIMD (Single Instruction Multiple Data) della CPU, se disponibili, per eseguire la stessa

operazione (moltiplicazione e accumulo) su più dati contemporaneamente all'interno di una singola iterazione del ciclo interno o su più iterazioni raggruppate.

L'uso combinato di queste due tecniche di parallelizzazione consente di sfruttare sia il parallelismo a livello di core (distribuendo i blocchi tra i thread) sia il parallelismo a livello di dati all'interno di ciascun core (usando SIMD per i calcoli sulle righe). Questo approccio è particolarmente vantaggioso per matrici di grandi dimensioni.

```
1: function OPENMPMULTIPLYHLL(struct MatriceHLL * mat, struct Vector * vec,  
    struct Vector * result)  
2:   for b  $\leftarrow$  0 to mat  $\rightarrow$  numBlocks - 1 do  
3:     blocco  $\leftarrow$  mat  $\rightarrow$  blocks[b]  
4:     inizioRigaGlobale  $\leftarrow$  b  $\times$  mat  $\rightarrow$  HackSize  
5:     #pragma omp parallel for schedule(static)  
6:     maxNZ  $\leftarrow$  blocco  $\rightarrow$  MAXNZ  
7:     for i  $\leftarrow$  0 to blocco  $\rightarrow$  M - 1 do  
8:       t  $\leftarrow$  0.0  
9:       inizioRiga  $\leftarrow$  i  $\times$  maxNZ  
10:      #pragma omp simd            $\triangleright$  Parallelizza il calcolo della somma per la riga i  
11:      for j  $\leftarrow$  0 to maxNZ - 1 do   $\triangleright$  Nota: L'accesso a vec[JA[...]] potrebbe non  
        essere ottimale per SIMD se gli indici JA non sono contigui.  
12:        t  $\leftarrow$  t + blocco  $\rightarrow$  AS[inizioRiga+j]  $\times$  vec  $\rightarrow$  data[blocco  $\rightarrow$  JA[inizioRiga+j]]  
13:      end for  
14:      result  $\rightarrow$  data[inizioRigaGlobale + i]  $\leftarrow$  t  
15:    end for  
16:  end for  
17: end function
```

## Ottimizzazione del layout dati per la moltiplicazione

Poiché l'approccio iniziale memorizzava gli elementi non nulli all'interno dei blocchi in formato *row-major*, è stata implementata una seconda versione che utilizza un ordinamento *column-major*.

Questa scelta consente di migliorare la parallelizzazione su due livelli:

- **Parallelizzazione a livello di blocco**, mediante `#pragma omp parallel for`, assegnando blocchi indipendenti ai vari thread.
- **Vettorizzazione SIMD sul ciclo più interno**, grazie al fatto che gli elementi appartenenti alla stessa colonna (all'interno del blocco) sono memorizzati contigui in memoria, facilitando l'utilizzo dei registri vettoriali senza operazioni di mascheramento o remainder.

In questo modo si ottiene un miglior utilizzo dei registri SIMD, una riduzione dell'overhead di sincronizzazione e un aumento generale delle prestazioni della moltiplicazione matrice-vettore.

```
1: function OPENMPMULTIPLYHLLFLAT(struct FlatELLMatrix * mat, struct Vector  
    * vec, struct Vector * result)  
2:   hackSize  $\leftarrow$  mat  $\rightarrow$  hack  
3:   #pragma omp parallel for schedule(static)            $\triangleright$  Parallelizza sui blocchi
```

```

4:   for block ← 0 to mat → numBlocks – 1 do
5:     block_start ← mat → block_offsets[block]
6:     maxNZ ← mat → block_nnz[block]
7:     rowNumber ← mat → block_rows[block]
8:     for j ← 0 to maxNZ – 1 do
9:       flat_idx ← block_start + j × rowNumber
10:      #pragma omp simd           ▷ Vettorizzazione su tutte le righe del blocco
11:      for i ← 0 to rowNumber – 1 do
12:        col ← mat → col_indices_flat[flat_idx + i]
13:        val ← mat → values_flat[flat_idx + i]
14:        molt ← vec → vettore[col]
15:        result → vettore[hackSize × block + i] += val × molt
16:      end for
17:    end for
18:  end for
19: end function

```

## Ottimizzazione V2: Scheduling adattivo

Nella versione **V2** dell'algoritmo di moltiplicazione è stato introdotto uno `schedule(guided)` all'interno della direttiva `#pragma omp parallel for`.

Questa scelta consente a OpenMP di distribuire dinamicamente il carico di lavoro tra i thread: inizialmente vengono assegnati blocchi più grandi, mentre successivamente i thread ricevono blocchi via via più piccoli. Tale strategia è particolarmente efficace nel caso di matrici sparse irregolari, dove il numero di elementi non nulli per blocco può variare, permettendo di bilanciare meglio il carico computazionale tra i thread.

### 2.2.4 Calcolo Parallelo con CUDA

Per poter procedere con il calcolo parallelo sulla GPU tramite CUDA, la matrice in formato HLL è stata convertita in un formato più adatto all'architettura della GPU, definito come `FlatELLMatrix`. Questa conversione mira a ottimizzare gli accessi alla memoria globale della GPU. La struttura dati è la seguente:

```

typedef struct FlatELLMatrix {
  double* values_flat;                      ▷ Valori flattenati (Blocco 0, Blocco 1, ...)
  int* col_indices_flat;                    ▷ Indici di colonna flattenati
  int* block_offsets;                       ▷ Offset inizio dati flat per ogni blocco
  int* block_nnz;                          ▷ Max non-zero per riga per blocco (MAXNZ per blocco)
  int* block_rows;                         ▷ Numero di righe per blocco
  int total_values;                        ▷ Totale elementi non-zero memorizzati
  int numBlocks;                           ▷ Numero blocchi
} FlatELLMatrix;

```

La conversione da HLL a FlatELL è realizzata nella funzione `convertHLLToFlatELL`, situata nel file `mat/cuda_luca.cu`. La struttura `FlatELLMatrix` memorizza tutti i valori (`values_flat`) e gli indici di colonna (`col_indices_flat`) dei blocchi HLL in due array contigui ("flattenati") in maniera column major. Gli array ausiliari (`block_offsets`,

`block_nnz`, `block_rows`) permettono di ricostruire la struttura logica dei blocchi all'interno dei kernel CUDA.

La scelta di convertire il formato nasce dall'esigenza di:

- **Ottimizzare l'accesso alla memoria globale della GPU:** Il formato flattenato favorisce accessi coalescenti alla memoria quando i thread di un warp accedono a locazioni contigue, riducendo le latenze.
- **Semplificare la logica del Kernel:** Un layout di memoria lineare è generalmente più semplice da gestire nei kernel CUDA rispetto a strutture dati complesse come array di puntatori a blocchi.
- **Massimizzare il throughput:** L'organizzazione compatta dei dati permette di sfruttare meglio la larghezza di banda della memoria della GPU.

Questo formato rende quindi l'operazione di prodotto matrice-vettore più adatta all'esecuzione su GPU. Di seguito vengono presentati tre diversi kernel CUDA implementati per eseguire il calcolo su questa struttura dati.

## Kernel 1 CUDA: Un Thread per Riga

Il primo kernel (`Kernel1`), implementato nella funzione `__global__ void matvec_flatell_kernel` nel file `mat/cuda_luca.cu`, adotta una strategia di parallelizzazione semplice e diretta: assegna **un thread CUDA per ogni riga globale** della matrice originale.

Ogni thread esegue i seguenti passi:

- Calcola l'indice della riga globale (`global_row`) che gli è stata assegnata in base al proprio ID (`blockIdx`, `threadIdx`).
- Determina l'ID del blocco (`block_id`) e l'indice della riga all'interno di quel blocco (`local_row`) a cui corrisponde la `global_row`.
- Utilizza gli array `block_offsets`, `block_nnz`, e `block_rows` della struttura `FlatELLMatrix` per trovare l'inizio dei dati del proprio blocco e il numero massimo di elementi non nulli per riga (`max_nnz`) in quel blocco.
- Itera da  $j = 0$  a  $\text{max\_nnz}-1$ , accedendo ai valori e agli indici di colonna corrispondenti alla propria riga (`local_row`) all'interno degli array flattenati (`values_flat`, `col_indices_flat`). L'accesso avviene calcolando un indice complesso basato su `block_start`, `j`, `rows_in_block`, e `local_row`.
- Controlla se l'indice di colonna (`col`) è valido (maggiore o uguale a 0, assumendo -1 per padding).
- Se l'indice è valido, moltiplica il valore della matrice per l'elemento corrispondente del vettore  $x$  (letto dalla memoria globale) e accumula il risultato in una somma parziale (`sum`).
- Infine, scrive il risultato finale (`sum`) nella posizione `global_row` del vettore di output  $y$ .

## Pseudocodice Kernel 1:

```
1: function MATVEC_FLATELL_KERNEL(dMat, x, y, numTotalRows)
2:   global_row  $\leftarrow$  blockIdx.x  $\times$  blockDim.x + threadIdx.x
3:   if global_row  $\geq$  numTotalRows then
4:     return
5:   end if
6:   block_id  $\leftarrow$  ...                                 $\triangleright$  Determine block containing global_row
7:   local_row  $\leftarrow$  ...                                 $\triangleright$  Determine local row index within the block
8:   sum  $\leftarrow$  0.0
9:   max_nnz  $\leftarrow$  dMat.block_nnz[block_id]       $\triangleright$  Max non-zeros per row in this block
10:  block_start  $\leftarrow$  dMat.block_offsets[block_id]       $\triangleright$  Start index for block data
11:  rows_in_block  $\leftarrow$  dMat.block_rows[block_id]       $\triangleright$  Number of rows in this block
12:  for j  $\leftarrow$  0 to max_nnz - 1 do
13:    flat_index  $\leftarrow$  block_start + j  $\times$  rows_in_block + local_row
14:    col  $\leftarrow$  dMat.col_indices_flat[flat_index]           $\triangleright$  Read column index
15:    if col  $\geq$  0 then                                 $\triangleright$  Check if it's a valid element (not padding)
16:      value  $\leftarrow$  dMat.values_flat[flat_index]           $\triangleright$  Read matrix value
17:      sum  $\leftarrow$  sum + value  $\times$  x[col]            $\triangleright$  Read from x and multiply-accumulate
18:    end if
19:  end for
20:  y[global_row]  $\leftarrow$  sum                                 $\triangleright$  Write final result
21: end function
```

## Kernel 2 CUDA: Utilizzo di Memoria Condivisa

Il secondo kernel (*Kernel2*), implementato in `__global__ void matvec_flatell_kernel_2`, tenta di migliorare le prestazioni del Kernel 1 riducendo gli accessi alla memoria globale per il vettore *x* attraverso l'uso della **memoria condivisa** (shared memory). La memoria condivisa è una piccola cache veloce on-chip accessibile dai thread all'interno dello stesso blocco CUDA.

La strategia è la seguente:

- Come nel Kernel 1, ogni thread calcola la propria *global\_row*, *block\_id*, e *local\_row*.
- Ogni blocco di thread carica collaborativamente una porzione del vettore *x* nella memoria condivisa. L'idea è che se più thread nello stesso blocco accedono agli stessi elementi di *x*, questi possono essere letti una sola volta dalla memoria globale e poi letti più volte dalla veloce memoria condivisa. La porzione di *x* da caricare dipende dalla strategia scelta (es. caricare gli elementi corrispondenti agli ID dei thread nel blocco).
- È necessaria una sincronizzazione (`__syncthreads()`) dopo il caricamento per assicurarsi che tutti i dati siano disponibili in memoria condivisa prima che i thread inizino i calcoli.
- Durante il calcolo del prodotto riga-vettore, quando un thread necessita di un elemento *x*[*col*], controlla prima se quell'elemento è presente nella porzione di *x* caricata in memoria condivisa.

- Se l'elemento è in memoria condivisa, viene letto da lì (accesso veloce).
- Altrimenti, viene letto dalla memoria globale (accesso più lento).
- Il risultato finale viene accumulato e scritto nel vettore  $y$  come nel Kernel 1.

L'efficacia di questo approccio dipende fortemente dal pattern di accesso agli elementi di  $x$ . Se i thread di un blocco accedono a elementi di  $x$  molto sparsi, il beneficio della memoria condivisa potrebbe essere limitato o nullo, e potrebbe persino introdurre overhead.

## Pseudocodice Kernel 2:

```
1: function MATVEC_FLATELL_KERNEL_2(dMat, x, y, numTotalRows)
2:   _shared_ double shared_x[DIMENSIONE_SHARED_X]
3:   global_row  $\leftarrow \dots$ 
4:   if global_row  $\geq$  numTotalRows then
5:     return
6:   end if
7:   block_id  $\leftarrow \dots$ 
8:   local_row  $\leftarrow \dots$ 
9:   indice_x_da_caricare  $\leftarrow \dots$  ▷ Caricamento collaborativo di una porzione di x in shared_x
10:  if indice_x_da_caricare  $<$  dimensione_vettore_x then
11:    shared_x[threadIdx.x]  $\leftarrow x[iindice\_x\_da\_caricare]$  ▷ Accesso Globale
12:  end if
13:  _syncthreads() ▷ Sincronizza i thread del blocco
▷ Calcola prodotto riga-vettore
14:  sum  $\leftarrow 0.0$ 
15:  max_nnz  $\leftarrow \dots$ 
16:  block_start  $\leftarrow \dots$ 
17:  rows_in_block  $\leftarrow \dots$ 
18:  for j  $\leftarrow 0$  to max_nnz - 1 do
19:    flat_index  $\leftarrow \dots$ 
20:    col  $\leftarrow dMat.col\_indices\_flat[flat\_index]$ 
21:    if col  $\geq 0$  then
22:      value  $\leftarrow dMat.values\_flat[flat\_index]$ 
23:      x_val  $\leftarrow 0.0$  ▷ Leggi x da shared memory se possibile, altrimenti da globale
▷ Logica per determinare se col è nell'intervallo caricato in shared_x
24:      if col è in shared_x then
25:        x_val  $\leftarrow shared\_x[iindice\_in\_shared\_x]$  ▷ Accesso Condiviso
26:      else
27:        x_val  $\leftarrow x[col]$  ▷ Accesso Globale
28:      end if
29:      sum  $\leftarrow sum + value \times x\_val$ 
30:    end if
31:  end for
32:  y[global_row]  $\leftarrow sum$  ▷ Scrivi il risultato
33: end function
```

## Kernel 3 CUDA: Warp per Blocco, Lane per Riga

Il terzo kernel, implementato nella funzione `matvec_flatell_kernel_warpColonne`, adotta una strategia di parallelizzazione più fine, il kernel assegna un warp per ciascun blocco della matrice FlatELL. All'interno della warp, ogni thread elabora una riga locale del blocco, calcolando il prodotto scalare con il vettore  $x$ .

Grazie alla memorizzazione column-major, gli accessi a memoria sono coalesced, migliorando l'efficienza. Ogni thread scrive in una posizione distinta del vettore  $y$ , evitando la necessità di operazioni atomiche.

Questa variante migliora l'utilizzo della memoria globale sfruttando appieno la struttura SIMT della GPU, evitando branching e permettendo una saturazione più efficiente della banda di memoria.

```
1: function MATVEC_FLATELL_KERNEL_WARP_COLONNE(FlatELLMATRIX * dMat, double
   * x, double * y, int hack_size, int total_blocks)
2:   thread_id ← blockIdx.x * blockDim.x + threadIdx.x
3:   warp_id ← thread_id ≈ 5           ▷ Identifica il warp (thread_id diviso 32)
4:   lane ← thread_id & 31            ▷ Identifica la lane all'interno del warp
5:   if warp_id ≥ total_blocks then return
6:   end if
7:   row ← dMat->block_rows[warp_id]
8:   if row ≤ lane then return ▷ Lane inattive se eccedono il numero di righe nel blocco
9:   end if
10:  block_start ← dMat->block_offsets[warp_id]
11:  max_nnz_per_row ← dMat->block_nnz[warp_id]
12:  sum ← 0.0
13:  for j ← 0 to max_nnz_per_row - 2 do
14:    flat_idx ← block_start + j × row + lane
15:    col ← dMat->col_indices_flat[flat_idx]
16:    val ← dMat->values_flat[flat_idx]
17:    molt ← x[col]
18:    sum += val × molt
19:  end for
20:  flat_idx ← block_start + (max_nnz_per_row - 1) × row + lane
21:  col ← dMat->col_indices_flat[flat_idx]
22:  val ← dMat->values_flat[flat_idx]
23:  molt ← x[col]
24:  sum += val × molt
25:  y[hack_size * warp_id + lane] ← sum
26: end function
```

La funzione intrinseca CUDA `__shfl_down_sync(mask, var, delta, width=warpSize)` è un'operazione di shuffle a livello di warp. Permette a un thread di leggere direttamente il valore della variabile `var` da un altro thread nello stesso warp, identificato da un offset `delta` verso il "basso" (ID del thread sorgente = ID thread corrente + delta).

## 2.3 Implementazione del Formato CSR per Matrici Sparse

In questa sezione viene presentata l'implementazione del formato CSR per memorizzazione efficiente di matrici sparse, non mi soffermerò troppo sul funzionamento di algoritmi e non approfondirò tutte le terminologia usate soprattutto se già citate precedentemente.

### 2.3.1 Descrizione breve del formato CSR

Il formato CSR (Compressed Sparse Row) è uno schema di memorizzazione per matrici che mira a ridurre l'occupazione di memoria memorizzando solamente gli elementi non nulli e le informazioni necessarie per ricostruire la loro posizione. La struttura dati utilizzata per rappresentare una matrice in formato CSR è la seguente:

```
typedef struct MatriceCsr {
    unsigned int width, height;                                ▷ Dimensioni della matrice
    unsigned int *iRP;                                         ▷ Array di dim. height+1
    unsigned int *jValori;                                      ▷ Array di dim. nz
    double *valori;                                           ▷ Non-zero Values - Array di dim. nz
} MatriceCsr;
```

Questo formato utilizza tre vettori principali per rappresentare la matrice:

- **valori**: Un array (di dimensione **nz**) che contiene i valori numerici degli elementi non nulli della matrice, memorizzati per righe (prima tutti gli elementi della riga 0, poi quelli della riga 1, e così via).
- **jValori**: Un array (di dimensione **nz**) che memorizza gli indici di colonna corrispondenti a ciascun elemento presente nell'array **valori**. Quindi, **jValori[k]** è l'indice di colonna del valore memorizzato in **valori[k]**.
- **iRP** (Integer Row Pointers): Un array (di dimensione **height + 1**) che permette di individuare l'inizio e la fine degli elementi di ciascuna riga all'interno degli array **valori** e **jValori**. In particolare, gli elementi della riga *i* (con *i* che va da 0 a **height-1**) si trovano negli indici compresi tra **iRP[i]** (incluso) e **iRP[i+1]** (escluso) degli array **valori** e **jValori**. L'ultimo elemento, **iRP[height]**, contiene sempre il valore **nz**, questo per far sì che l'ultima riga abbiamo un delimitatore di fine.

### 2.3.2 Calcolo Seriale

Per il calcolo seriale della moltiplicazione matrice-vettore, l'algoritmo è relativamente diretto. Si procede iterando attraverso tutte le righe della matrice, una dopo l'altra. Per ciascuna riga, si effettua il calcolo della moltiplicazione riga per colonna (prodotto scalare) tra gli elementi non nulli di quella specifica riga della matrice e gli elementi corrispondenti del vettore di input.

Dal punto di vista implementativo, ciò richiede di scorrere tutti gli elementi memorizzati nel vettore dei valori (**valori**). Di conseguenza, l'algoritmo deve necessariamente visitare ogni elemento non-zero della matrice almeno una volta. Pertanto, la complessità computazionale dell'algoritmo seriale è lineare rispetto al numero totale di elementi non-zero della matrice, e si esprime come  $O(NZ)$ , dove  $NZ$  rappresenta il numero totale di elementi non-zero.

```
1: function SERIALCSRMULT(csr, vec, result)
2:   nrows  $\leftarrow$  vec.righe
3:   for i  $\leftarrow 0$  to nrows – 1 do
4:     sum  $\leftarrow$  0.0
5:     for j  $\leftarrow$  csr.iRP[i] to csr.iRP[i + 1] – 1 do
6:       sum  $\leftarrow$  sum + csr.valori[j]  $\times$  vec.vettore[csr.jValori[j]]
7:     end for
8:     result.vettore[i]  $\leftarrow$  sum
9:   end for
10:  end function
```

### 2.3.3 Calcolo Parallelo con OpenMP

Per la versione parallelizzata della moltiplicazione matrice-vettore CSR tramite OpenMP, si è partiti dalla struttura del codice seriale, introducendo le direttive OpenMP per distribuire il lavoro sui thread disponibili. La parallelizzazione principale si applica al ciclo **for** esterno, che itera sulle righe della matrice *i*. La direttiva utilizzata è `#pragma omp parallel for schedule(static)`. La scelta dello scheduler **static** è derivata dalle seguenti considerazioni:

1. **Osservazione Euristica:** Dai test effettuati, si è notato che, per il nostro specifico dataset e ambiente, le prestazioni ottenute con `schedule(static)` erano generalmente buone e non significativamente inferiori rispetto a quelle ottenibili con scheduler dinamici come `guided` o `dynamic`.
2. **Caratteristiche delle Matrici e Semplicità:** Una porzione considerevole delle matrici nel dataset di test presenta una distribuzione degli elementi non-zero relativamente bilanciata tra le righe, o una sparsità concentrata sulla diagonale. Queste caratteristiche tendono a mitigare i problemi di squilibrio di carico (load imbalance) che sono il principale svantaggio dello scheduling statico. Data questa osservazione, e considerando la semplicità e il minor overhead intrinseco di **static**, si è scelto di adottarlo come approccio predefinito, visti i risultati complessivamente soddisfacenti ottenuti. Si riconosce che per matrici con forte irregolarità nella distribuzione dei non-zeri, altri scheduler potrebbero essere preferibili.

Relativamente al ciclo **for** interno (indice *j*), che esegue il calcolo del prodotto scalare per la singola riga, è stata introdotta la direttiva `#pragma omp simd reduction(+:sum)`. L'intento è di sfruttare la vettorizzazione SIMD. Le prove sperimentali suggeriscono che questa direttiva apporta un contributo positivo, seppur talvolta minimo, alle performance. È importante sottolineare che l'efficacia della clausola `simd`, specialmente in presenza degli accessi indiretti alla memoria, dipende fortemente dalla versione del compilatore, dall'implementazione della libreria OpenMP e dall'architettura specifica della CPU. Non essendo stato possibile effettuare test approfonditi su una vasta gamma di ambienti, la direttiva è stata mantenuta nell'implementazione finale utilizzata per i test sul server, dove ha dimostrato di fornire risultati complessivamente validi. In un'analisi di complessità parallela molto semplificata, se assumiamo di poter processare *n* righe della matrice simultaneamente, la complessità ideale target si riduce a  $O(NZ/n)$ , dove *NZ* è il numero totale di elementi non-zero. Questa è una stima grossolana che offre un'idea del limite teorico a cui aspirare, che non tiene conto di limiti hardware e che assume *nz* distribuiti uniformemente tra righe.

```
1: function PARALLELCSRMULT(csr, vec, result)
```

```
2:      nrows ← vec.righe
3:      #pragma omp parallel for schedule(static)
4:      for i ← 0 to nrows - 1 do
5:          sum ← 0.0
6:          #pragma omp simd reduction(+:sum)
7:          for j ← csr.iRP[i] to csr.iRP[i + 1] - 1 do
8:              sum ← sum + csr.valori[j] × vec.vettore[csr.jValori[j]]
9:          end for
10:         result.vettore[i] ← sum
11:     end for
12: end function
```

### 2.3.4 Calcolo Parallelo con CUDA

Per quanto riguarda CUDA, ogni kernel è stato progettato in modo tale da poter essere eseguito con un numero variabile di threads per blocco e quindi testare diverse configurazioni.

#### kernel1

Il primo kernel CUDA adotta un approccio di parallelizzazione diretto e semplice: si assegna a ciascun thread CUDA la responsabilità di calcolare il risultato per una singola riga della matrice. Ogni thread, quindi, itera sugli elementi non-zero della propria riga ed esegue il prodotto scalare con il vettore di input  $\mathbf{x}$ .

Nonostante la sua semplicità concettuale, questa prima versione ha mostrato risultati interessanti durante i test. Si è misurato un throughput computazionale pari a circa il **14%** del valore di picco teorico della GPU e un throughput di memoria effettivo pari a circa il **50%** della banda di memoria massima teorica. Pur essendo valori non ottimali, indicano che anche questa implementazione basilare riesce a sfruttare parte del parallelismo della GPU, superando in quasi tutti i casi testati le prestazioni ottenute con la versione parallelizzata tramite OpenMP.

```
1: function CSR_MATVEC_MUL(d_mat, d_vec, d_result)
2:     row ← blockIdx.x × blockDim.x + threadIdx.x
3:     if row < d_mat.height then
4:         sum ← 0.0
5:         for j ← d_mat.iRP[row] to d_mat.iRP[row + 1] - 1 do
6:             sum ← sum + d_mat.valori[j] × d_vec.vettore[d_mat.jValori[j]]
7:         end for
8:         d_result.vettore[row] ← sum
9:     end if
10: end function
```

#### kernel2

Il secondo approccio implementato mira a sfruttare meglio il parallelismo intrinseco della GPU, in particolare a livello di *warp*. In questo kernel, l'unità di lavoro fondamentale non è più il singolo thread, ma l'intero warp (composto da 32 thread): a ciascun warp viene assegnata la responsabilità di calcolare il risultato per una singola riga della matrice.

L'algoritmo procede come segue:

- I 32 thread del warp collaborano per processare gli elementi non-zero della riga assegnata. Ogni thread accede agli elementi con un passo di 32 (stride), garantendo accessi potenzialmente più coalescenti (specialmente agli array `valori` e `jValori` della matrice CSR).
- Ciascun thread calcola una somma parziale basata sugli elementi che ha processato.
- La gestione degli elementi rimanenti alla fine di una riga (quando il numero di non-zeri non è un multiplo di 32) viene ottimizzata: solo i thread attivi (con `lane id < remaining`) partecipano al calcolo, evitando lavoro inutile per gli altri thread del warp
- .
- Al termine del calcolo degli elementi della riga, viene eseguita una riduzione parallela delle somme parziali accumulate da ciascun thread all'interno del warp. Questa operazione viene implementata efficientemente utilizzando primitive CUDA come `--shfl_down_sync`.
- Infine, il primo thread del warp (lane 0), che ora detiene la somma totale per la riga, scrive il risultato finale nella posizione appropriata del vettore `y`.

Dal punto di vista delle performance, questo kernel ha mostrato un netto miglioramento rispetto alla versione base. Si è raggiunto un throughput computazionale pari a circa il **76%** del picco teorico e un throughput di memoria effettivo molto elevato, circa il **92%** della banda massima. L'analisi del profiler indica anche un buon tasso di occupazione dei multiprocessori (circa **90%**).

Nonostante l'elevata efficienza nell'uso della banda di memoria (92%) e la buona occupazione, il throughput computazionale si ferma al 76%. Questo suggerisce che, sebbene la banda sia sfruttata bene *quando* si accede alla memoria, potrebbero esserci ancora colli di bottiglia legati alla latenza della memoria o alla contesa tra diversi warp che tentano di accedere simultaneamente alla memoria globale (in particolare per gli accessi indiretti al vettore `x`). Questi fattori possono portare i thread ad attendere i dati (*stalls*), limitando così il throughput computazionale effettivo nonostante l'hardware sia mantenuto relativamente occupato.

```
1: function WARPREDUCESUM(val)
2:   offset  $\leftarrow$  16
3:   while offset  $>$  0 do
4:     val  $\leftarrow$  val + __shfl_down_sync(0xFFFFFFFF, val, offset)
5:     offset  $\leftarrow$  offset/2
6:   end while
7:   return val
8: end function

1: function CRS_MAT_32_WAY(d_mat, d_vec, d_result)
2:   id  $\leftarrow$  blockIdx.x  $\times$  blockDim.x + threadIdx.x
3:   realRow  $\leftarrow$  id/32
4:   position  $\leftarrow$  id (mod 32)
5:   if realRow  $\geq$  d_mat.height then
6:     return
7:   end if
8:   base  $\leftarrow$  d_mat.iRP[realRow]
9:   rowDim  $\leftarrow$  d_mat.iRP[realRow + 1] - base
10:  sum  $\leftarrow$  0.0
11:  num_full_warps  $\leftarrow$  rowDim/32
12:  for i  $\leftarrow$  0 to num_full_warps - 1 do
13:    idx  $\leftarrow$  base + i  $\times$  32 + position
14:    col_index  $\leftarrow$  d_mat.jValori[idx]
15:    matVal  $\leftarrow$  d_mat.valori[idx]
16:    vectVal  $\leftarrow$  d_vec.vettore[col_index]
17:    sum  $\leftarrow$  sum + matVal  $\times$  vectVal
18:  end for
19:  remaining  $\leftarrow$  rowDim (mod 32)
20:  if remaining  $>$  0 then
21:    start_of_remaining  $\leftarrow$  base + rowDim - remaining
22:    if position  $<$  remaining then
23:      idx  $\leftarrow$  start_of_remaining + position
24:      col_index  $\leftarrow$  d_mat.jValori[idx]
25:      matVal  $\leftarrow$  d_mat.valori[idx]
26:      vectVal  $\leftarrow$  d_vec.vettore[col_index]
27:      sum  $\leftarrow$  sum + matVal  $\times$  vectVal
28:    end if
29:  end if
30:  sum  $\leftarrow$  warpReduceSum(sum)
31:  if position = 0 then
32:    d_result.vettore[realRow]  $\leftarrow$  sum
33:  end if
34: end function
```

### kernel3

Per questo terzo kernel, si è deciso di mantenere la struttura logica del kernel precedente (warp-per-riga con riduzione tramite shuffle), focalizzando invece l'ottimizzazione sulla *struttura dati della matrice* per migliorare l'efficienza degli accessi alla memoria globale.

La modifica principale consiste nell'introduzione di *padding* all'interno della rappresentazione della matrice. L'obiettivo è di allineare i dati in memoria in modo tale da massimizzare la *coalescenza* degli accessi da parte dei thread di un warp. In altre parole, aggiungendo strategicamente elementi fintizi, si cerca di far sì che le letture e le scritture effettuate possano essere raggruppate dal memory controller della GPU nel minor numero possibile di transazioni di memoria. Viene inoltre menzionato un tentativo, nella parte iniziale del kernel, di utilizzare tipi vettoriali CUDA (come `int2`) per caricare gli indici di riga (`iRP`) in modo più efficiente.

Questa strategia di modifica della struttura dati per favorire gli accessi si è rivelata efficace dal punto di vista della memoria: il throughput di memoria (*memory throughput*) misurato è salito al **95%** della banda massima teorica, indicando un utilizzo molto efficiente del bus di memoria.

Tuttavia, l'ottimizzazione degli accessi alla memoria ha comportato un impatto negativo sul lato computazionale. Il throughput computazionale (*compute throughput*) è diminuito rispetto al kernel precedente, scendendo al **65%**. Questo suggerisce che l'overhead introdotto dal padding – sia esso dovuto alla necessità di leggere/processare più dati, a una maggiore complessità nell'indicizzazione, o ad altri fattori correlati – ha controbilanciato negativamente i benefici ottenuti dalla migliore efficienza della memoria. Si tratta di un classico esempio di trade-off tra ottimizzazioni orientate alla memoria e l'impatto sul carico computazionale effettivo.

```
1: function CRS_MAT_32_WAY_COAL(d_mat, d_vec, d_result)
2:   id  $\leftarrow$  blockIdx.x  $\times$  blockDim.x + threadIdx.x
3:   realRow  $\leftarrow$  id/32            $\triangleright$  Riga processata da questo warp (divisione intera)
4:   loaded_ints  $\leftarrow$  LoadInt2(d_mat.iRP, realRow)
5:   base  $\leftarrow$  loaded_ints.x           $\triangleright$  Indice inizio elementi riga
6:   end_ptr  $\leftarrow$  loaded_ints.y       $\triangleright$  Indice fine elementi riga (+1)
7:   ...tha same as before....
8: end function
```

#### 2.3.4.1 Gestione del Warp: Shuffle e Riduzione Diretta

Dopo vari tentativi e un'attenta analisi delle matrici, ho deciso di proseguire con l'approccio della **suddivisione del lavoro per warp** per ottimizzare la sincronizzazione. Ho introdotto una modifica significativa: **dividere ulteriormente il lavoro assegnato a ciascun warp in  $N$  sezioni indipendenti**, creando di fatto dei "mini-warp" all'interno del warp principale.

Questa strategia non complica eccessivamente la gestione dei branching dei warp e consente di **sfruttare meglio i thread** nei casi in cui il carico di lavoro (le righe della matrice) non è bilanciato.

Ho sperimentato diverse dimensioni per questi mini-warp e, per la loro sincronizzazione interna finalizzata al risultato, ho adottato due approcci principali:

1. **Shuffle Down Sync**: Utilizzo delle **maschere di bit** per sincronizzare il lavoro locale di ogni mini-warp.
2. **Shared Memory**: Ho impiegato la **memoria condivisa** per effettuare una **riduzione top-down classica** per ciascun mini-warp.

## Warp Custom Reduction

Ad ogni lancio di un blocco, viene allocata un'area di **shared memory** la cui dimensione è pari al numero di thread nel blocco moltiplicato per la dimensione di un float.

Ogni thread determina il proprio **mini-warp** di appartenenza e la sua posizione al suo interno. Successivamente, ciascun mini-warp elabora  $N$  elementi della riga per volta, dove  $N$  rappresenta la dimensione del mini-warp. L'ultimo ciclo, invece, esegue un “unroll” per elaborare solo i valori rimanenti.

A questo punto, ogni thread scrive in shared memory il valore calcolato. Segue una fase di sincronizzazione del warp, dopodiché viene eseguita una **riduzione in parallelo** tra i vari mini-warp, che operano su dati indipendenti.

Al termine della sincronizzazione, il thread con l'ID di mini-warp pari a zero (ovvero il “master” thread di ogni mini-warp) scrive il risultato complessivo del suo mini-warp nel vettore finale.

```
1: function CRS_MAT_MINIWARP(d_mat, d_vec, d_result, miniWarpSize)
2:   Dichiarazione: reduceVals in memoria condivisa
3:   id  $\leftarrow$  blockIdx.x  $\times$  blockDim.x + threadIdx.x
4:   realRow  $\leftarrow$  id / miniWarpSize
5:   position  $\leftarrow$  id (mod miniWarpSize)
6:   if realRow  $\geq$  d_mat.height then
7:     return
8:   end if
9:   base  $\leftarrow$  d_mat.iRP[realRow]
10:  rowDim  $\leftarrow$  d_mat.iRP[realRow + 1] - base
11:  sum  $\leftarrow$  0.0
12:  for i  $\leftarrow$  0 to (rowDim/miniWarpSize) - 1 do
13:    idx  $\leftarrow$  base + i  $\times$  miniWarpSize + position
14:    col_index  $\leftarrow$  d_mat.jValori[idx]
15:    matVal  $\leftarrow$  d_mat.valori[idx]
16:    vectVal  $\leftarrow$  d_vec.vettore[col_index]
17:    sum  $\leftarrow$  sum + matVal  $\times$  vectVal
18:  end for
19:  remaining  $\leftarrow$  rowDim (mod miniWarpSize)
20:  if remaining > 0 then
21:    start_of_remaining  $\leftarrow$  base + rowDim - remaining
22:    if position < remaining then
23:      idx  $\leftarrow$  start_of_remaining + position
24:      col_index  $\leftarrow$  d_mat.jValori[idx]
25:      matVal  $\leftarrow$  d_mat.valori[idx]
26:      vectVal  $\leftarrow$  d_vec.vettore[col_index]
27:      sum  $\leftarrow$  sum + matVal  $\times$  vectVal
28:    end if
29:  end if
30:  reduceVals[threadIdx.x]  $\leftarrow$  sum
31:  _SYNCTHREADS
32:  for s  $\leftarrow$  miniWarpSize/2 down to 1 do
33:    if position < s then
34:      reduceVals[threadIdx.x]  $\leftarrow$  reduceVals[threadIdx.x] + reduceVals[threadIdx.x + s]
35:    end if
36:    _SYNCTHREADS
37:  end for
38:  if position = 0 then
39:    d_result.vettore[realRow]  $\leftarrow$  reduceVals[threadIdx.x]
40:  end if
41: end function
```

## Warp Custom Shuffle

La logica di divisione del lavoro e l'organizzazione in mini-warp sono identiche all'approccio precedente. Tuttavia, in questo caso **non viene istanziata alcuna memoria condivisa** ('shared memory'); il lavoro parziale viene invece mantenuto in una variabile locale a ciascun thread.

Una volta completate le operazioni di moltiplicazione e somma, viene calcolata la **bit-mask** specifica per il mini-warp corrente. Tramite la funzione ‘`__shfl_down_sync`’, viene eseguita la **riduzione della somma** all’interno di ogni mini-warp. Infine, il thread con l’ID di mini-warp pari a zero (il “master” di ciascun mini-warp) memorizza il risultato finale nel vettore dei risultati.

```
1: function CRS_MAT_MINIWARPWITHSHFL(d_mat, d_vec, d_result, miniWarpSize)
2:   id  $\leftarrow$  blockIdx.x  $\times$  blockDim.x + threadIdx.x
3:   realRow  $\leftarrow$  id / miniWarpSize
4:   position  $\leftarrow$  id (mod miniWarpSize)
5:   if realRow  $\geq$  d_mat.height then
6:     return
7:   end if
8:   base  $\leftarrow$  d_mat.iRP[realRow]
9:   rowDim  $\leftarrow$  d_mat.iRP[realRow + 1] – base
10:  sum  $\leftarrow$  0.0
11:  for i  $\leftarrow$  0 to (rowDim/miniWarpSize) – 1 do
12:    idx  $\leftarrow$  base + i  $\times$  miniWarpSize + position
13:    col_index  $\leftarrow$  d_mat.jValori[idx]
14:    matVal  $\leftarrow$  d_mat.valori[idx]
15:    vectVal  $\leftarrow$  d_vec.vettore[col_index]
16:    sum  $\leftarrow$  sum + matVal  $\times$  vectVal
17:  end for
18:  remaining  $\leftarrow$  rowDim (mod miniWarpSize)
19:  if remaining > 0 then
20:    start_of_remaining  $\leftarrow$  base + rowDim – remaining
21:    if position < remaining then
22:      idx  $\leftarrow$  start_of_remaining + position
23:      col_index  $\leftarrow$  d_mat.jValori[idx]
24:      matVal  $\leftarrow$  d_mat.valori[idx]
25:      vectVal  $\leftarrow$  d_vec.vettore[col_index]
26:      sum  $\leftarrow$  sum + matVal  $\times$  vectVal
27:    end if
28:  end if
29:  miniwarp_mask  $\leftarrow$  ((1u  $\ll$  miniWarpSize) – 1)
30:  miniwarp_mask  $\leftarrow$  miniwarp_mask  $\ll$  (((threadIdx.x (mod 32)) / miniWarpSize)  $\times$  miniWarpSize)
31:  for offset  $\leftarrow$  miniWarpSize/2 down to 1 do
32:    sum  $\leftarrow$  sum + __SHFL_DOWN_SYNC(miniwarp_mask, sum, offset)
33:  end for
34:  if position = 0 then
35:    d_result.vettore[realRow]  $\leftarrow$  sum
36:  end if
37: end function
```

# Capitolo 3

## Performance delle soulzioni

### 3.1 Spiegazione delle misurazioni

Le misurazioni sono state effettuate eseguendo 10 iterazioni per ciascuna configurazione, testando con 2, 4, 8, 16, 24, 30 e 40 thread. Per ogni configurazione sono stati calcolati la media e la varianza delle metriche ottenute per ciascuna matrice. Inoltre, è stato calcolato l'errore tra il vettore risultante  $y$  ottenuto dalla versione parallela e il vettore  $y_1$  ottenuto tramite l'operazione seriale.

Nel caso di OpenMP, il tempo di esecuzione è stato misurato utilizzando la funzione `omp_get_wtime()`. Mentre nel caso di CUDA la funzione `cudaEventCreate()`, `cudaEventRecord()`, `cudaEventSynchronzize()`,

Per quanto riguarda CUDA, i thread per blocco utilizzati sono stati 32, 64, 128 e 256. Tutte le misurazioni vengono salvate nel file `result/test.csv`.

La metrica di performance FLOPS è stata calcolata come segue:

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

dove  $NZ$  è il numero di elementi non nulli della matrice e  $T$  è il tempo di esecuzione in millisecondi.

### 3.2 Architettura su cui abbiamo testato

Le specifiche hardware del server del dipartimento utilizzato per i dati di questa relazione sono riassunte di seguito:

Componente	Specifiche
Processore	Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz
Core Fisici per Socket	10
Thread Logici per Socket	20 (con Hyper-Threading)
Numero di Socket	2
Core Fisici Totali	20
Thread Logici Totali	40
GPU	Nvidia RTX 5000

Tabella 3.1: Specifiche Hardware del Server di Acquisizione Dati

### 3.3 Analisi sui dati CPU

I test delle performance sono stati condotti variando il numero di thread per entrambi i formati **HLL** e **CSR**. Il massimo rendimento è stato raggiunto con 40 thread per il formato CSR, registrando un picco di **36.94 GFLOPS** sulla matrice **olafu.mtx**. Analogamente, con 40 thread per il formato HLL, si è ottenuto un picco di **25.83 GFLOPS** sulla matrice **af23560.mtx**. I risultati sottostanti riportati dai grafici derivano dal calcolo medio effettuato su 10 iterazioni per ciascuna configurazione di thread testata, al fine di garantire l'affidabilità delle misurazioni.

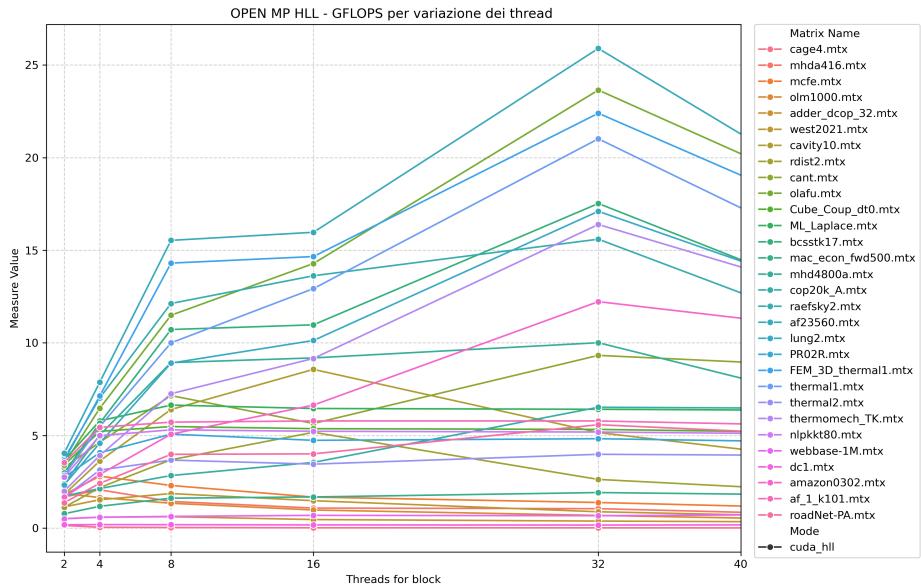


Figura 3.1: OPENMP HLL

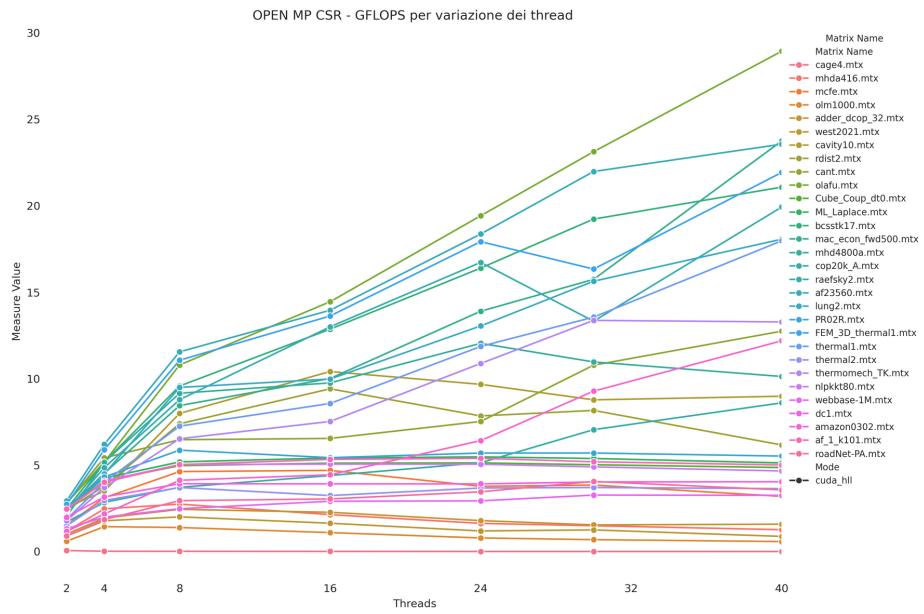


Figura 3.2: OPENMP CSR

Come si evince dal grafico, l'implementazione OpenMP basata sul formato CSR (barre blu) offre generalmente performance superiori rispetto all'implementazione HLL (barre arancioni) nella maggior parte delle matrici testate. Tuttavia, è possibile osservare alcune eccezioni significative: per matrici come `af23560.mtx`, `thermal2.mtx` e `webbase-1M.mtx`, il formato HLL raggiunge prestazioni comparabili o persino superiori al formato CSR. Questo comportamento può essere attribuito alle caratteristiche strutturali specifiche di queste matrici, che favoriscono l'accesso ai dati secondo il pattern implementato da HLL. È interessante notare come per matrici di grandi dimensioni come `nlpkkt80.mtx`, il formato CSR mostri un vantaggio prestazionale considerevole, raggiungendo il picco di performance dell'intero test. D'altra parte, per matrici con pattern di sparsità più irregolari, la struttura gerarchica di HLL sembra mitigare meglio gli effetti negativi degli accessi casuali alla memoria. Questi risultati evidenziano come la scelta del formato di memorizzazione ottimale per calcoli paralleli su matrici sparse dipenda fortemente dalle caratteristiche intrinseche della matrice e dal pattern di accesso ai dati che ne deriva.

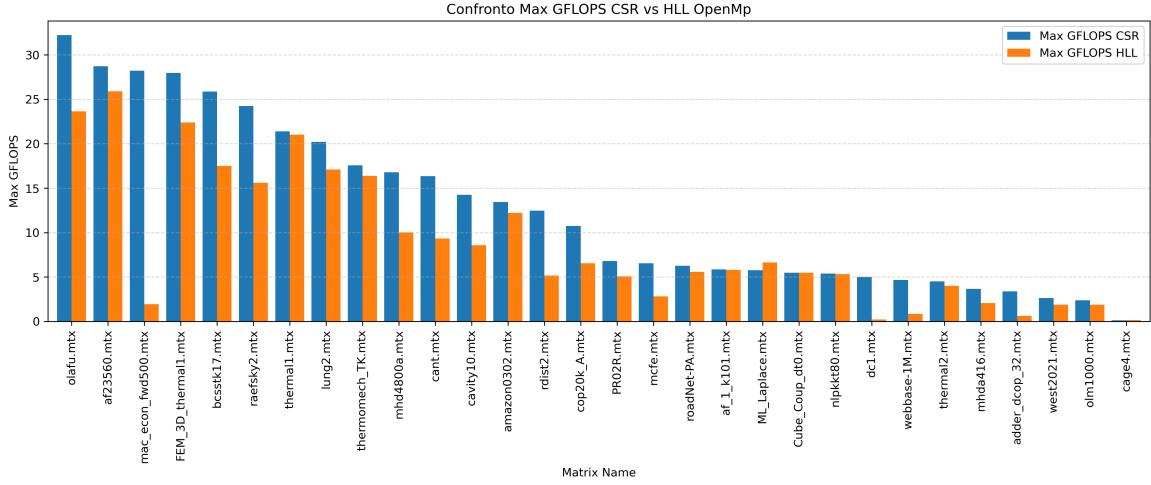


Figura 3.3: OPENMP HLL OPENMP CSR

### 3.4 Analisi sui dati GPU

Nel contesto delle implementazioni CUDA, è stata condotta un'analisi sistematica delle performance computazionali al variare del numero di thread per blocco per entrambi i formati di memorizzazione. I risultati evidenziano che l'implementazione ottimale ha raggiunto un picco prestazionale di **62.99826 GFLOPS** sulla matrice **MLaplace.mtx**, configurando **128 thread per blocco**. In confronto, il formato CSR ha registrato un massimo di **60.091 GFLOPS** sulla matrice **MLaplace.mtx**, con una configurazione di **256 thread per blocco**. È importante sottolineare che, per garantire l'affidabilità statistica delle misurazioni, tutti i dati presentati nei grafici rappresentano la media aritmetica di **10 iterazioni consecutive** per ciascuna configurazione di thread testata. Questa metodologia ha permesso di mitigare le fluttuazioni prestazionali dovute a fattori esterni, offrendo una valutazione robusta dell'efficienza computazionale delle diverse implementazioni.

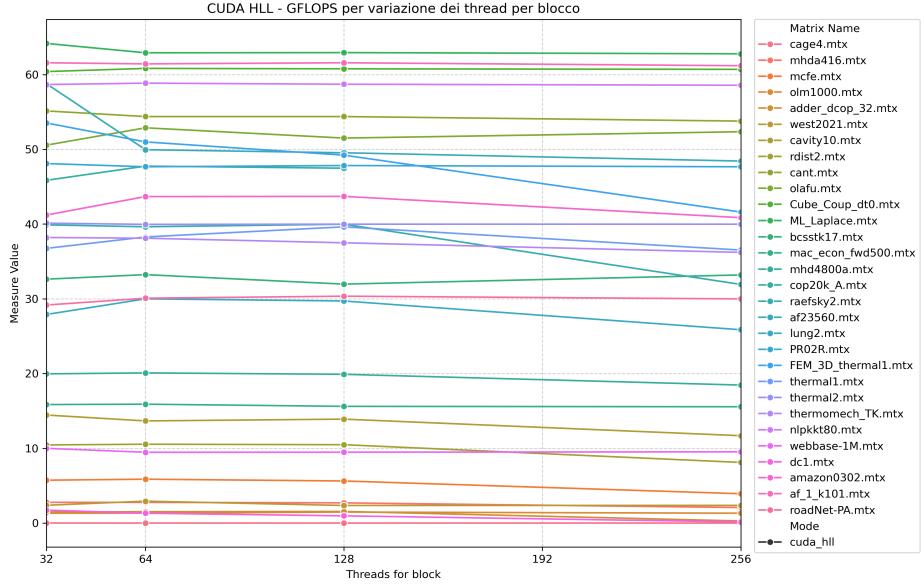


Figura 3.4: CUDA HLL

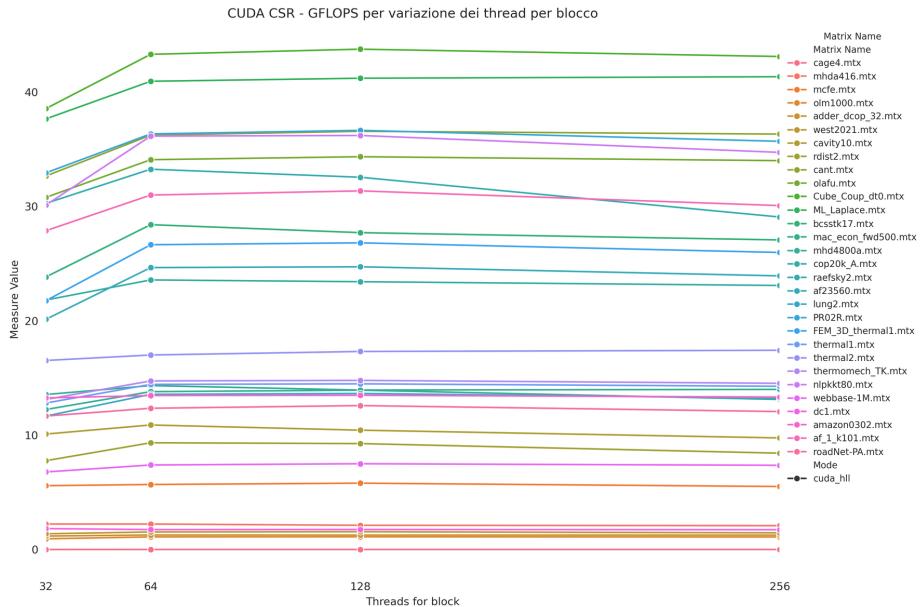


Figura 3.5: CUDA CSR

## Analisi delle prestazioni in funzione dei thread per blocco

Dai grafici si osserva che le prestazioni (GFLOPS) raggiungono il massimo con 64 o 128 thread per blocco, mentre diminuiscono con 256 thread per blocco. Questo avviene perché blocchi troppo grandi consumano più risorse (registri e memoria condivisa), riducendo il

numero di blocchi eseguibili in parallelo su ciascun multiprocessore e quindi l'efficienza complessiva. Inoltre, l'aumento dei thread per blocco comporta maggiori sincronizzazioni e può causare conflitti negli accessi alla memoria condivisa, specialmente con formati come l'HLL. Di conseguenza, 64 o 128 thread per blocco rappresentano un compromesso ottimale tra parallelismo ed efficienza nell'utilizzo delle risorse della GPU permettendo di massimizzare l'*occupancy*, ovvero il numero di thread realmente attivi in parallelo..

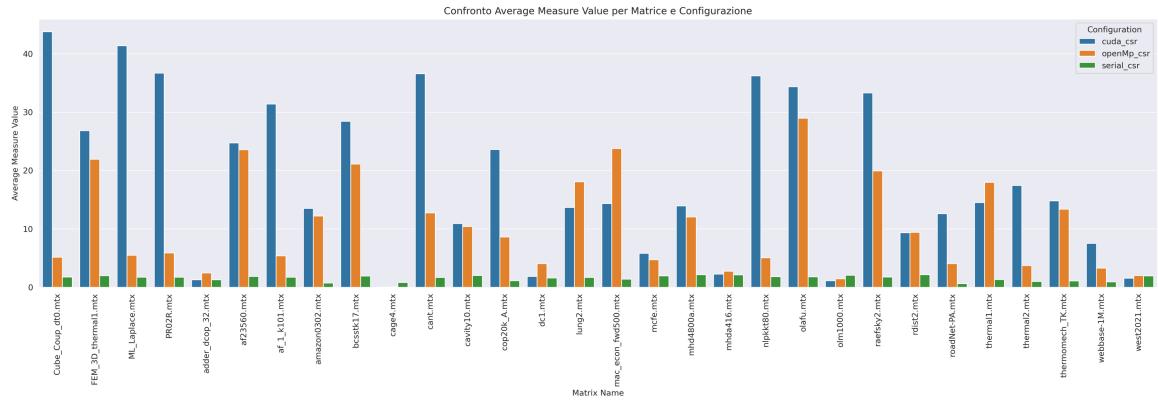


Figura 3.6: CSR

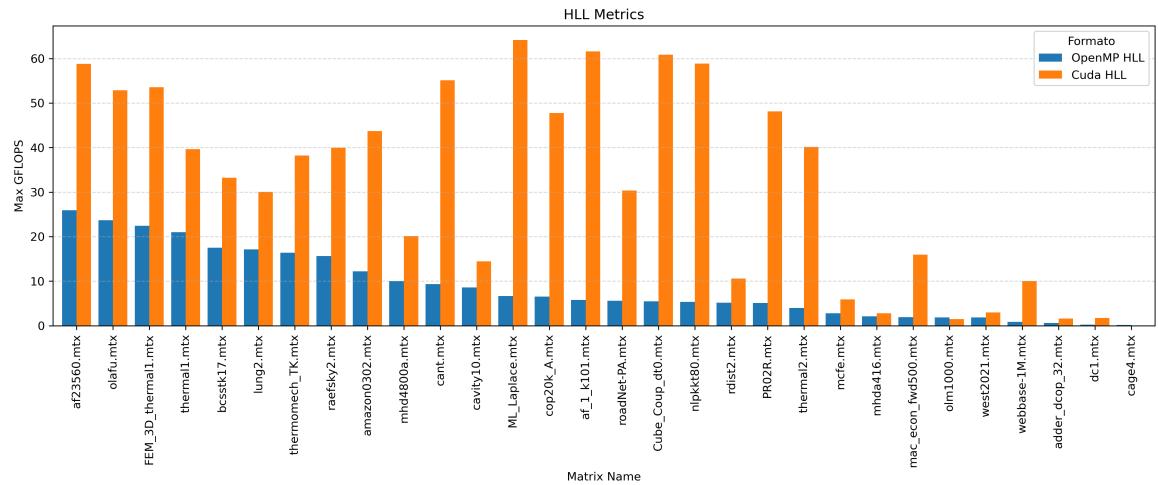


Figura 3.7: HLL

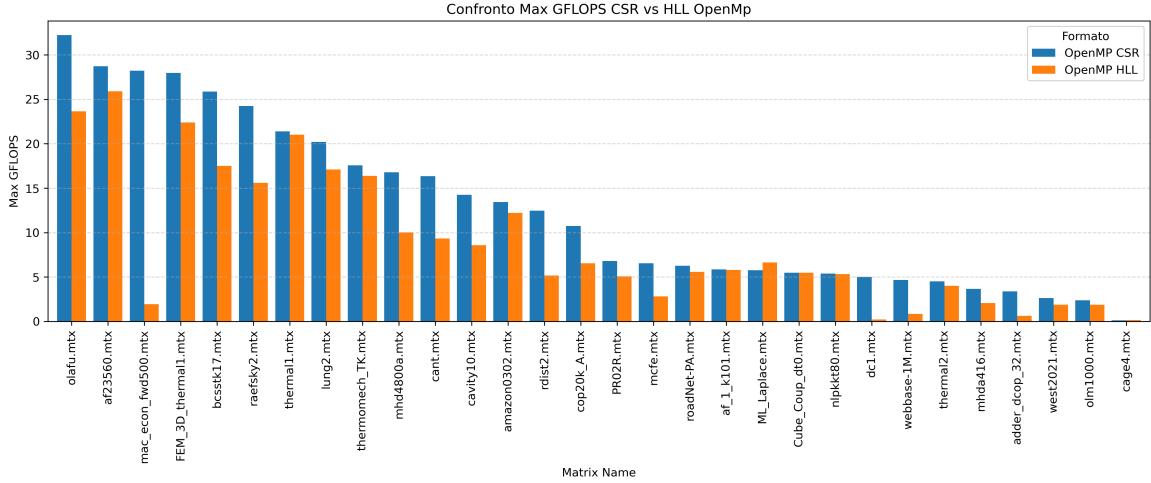


Figura 3.8: CUDA

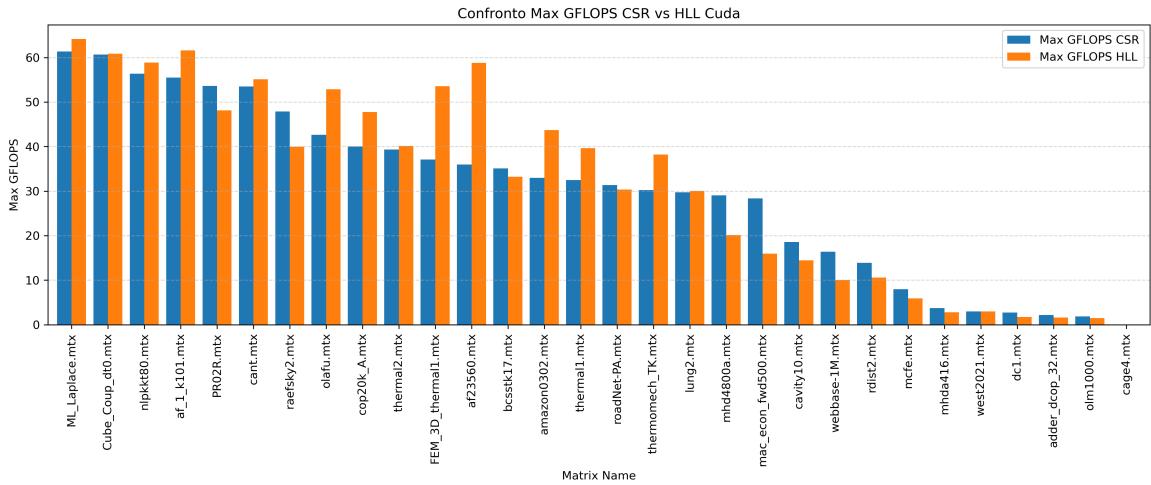


Figura 3.9: CUDA

Come si evince dal grafico, le implementazioni CUDA mostrano un comportamento prestazionale significativamente diverso rispetto a quanto osservato con OpenMP. Mentre nell'implementazione OpenMP il formato CSR risultava leggermente superiore, in ambiente CUDA assistiamo a un ribaltamento della situazione, con il formato HLL che raggiunge performance migliori nella maggioranza delle matrici testate. Questo fenomeno è particolarmente evidente per matrici come `ML_Laplace.mtx`, `af_1_k101.mtx` e `af23560.mtx`, dove HLL supera CSR con un margine considerevole. Tale inversione di tendenza è attribuibile alla fondamentale differenza architettonica tra CPU e GPU: l'architettura CUDA, con la sua massiccia parallelizzazione e i modelli di accesso alla memoria ottimizzati per pattern irregolari, riesce a sfruttare efficacemente la struttura gerarchica del formato HLL, compensando l'overhead che invece penalizzava questo formato nell'elaborazione su CPU multi-core con OpenMP.

### 3.5 Comparazione dei risultati

Confrontando i risultati ottenuti da CUDA e OpenMP, le prestazioni di CUDA risultano significativamente superiori grazie alla sua capacità di parallelizzare massicciamente il calcolo, in particolare nel formato HLL. L'efficienza della parallelizzazione si rivela maggiore in CUDA, soprattutto all'aumentare delle dimensioni delle matrici. Questo vantaggio è intrinseco alle GPU, che possono gestire migliaia di thread simultanei per operazioni omogenee, superando di gran lunga i 40 thread logici tipici delle CPU moderne.

Il formato HLL amplifica ulteriormente questo divario, ottimizzando l'accesso alla memoria e riducendo i colli di bottiglia computazionali. La scalabilità di CUDA si dimostra ottimale per problemi di grandi dimensioni, dove l'architettura a molti core delle GPU consente un'accelerazione quasi lineare. Al contrario, l'implementazione con OpenMP mostra limiti strutturali legati al parallelismo a grana più grossa e alla minore densità di thread gestibili dalla CPU.

In sintesi, CUDA rappresenta la soluzione più efficiente per algoritmi ad alta parallelizzazione, specialmente in contesti con matrici di grandi dimensioni e formati di dati ottimizzati come l'HLL.

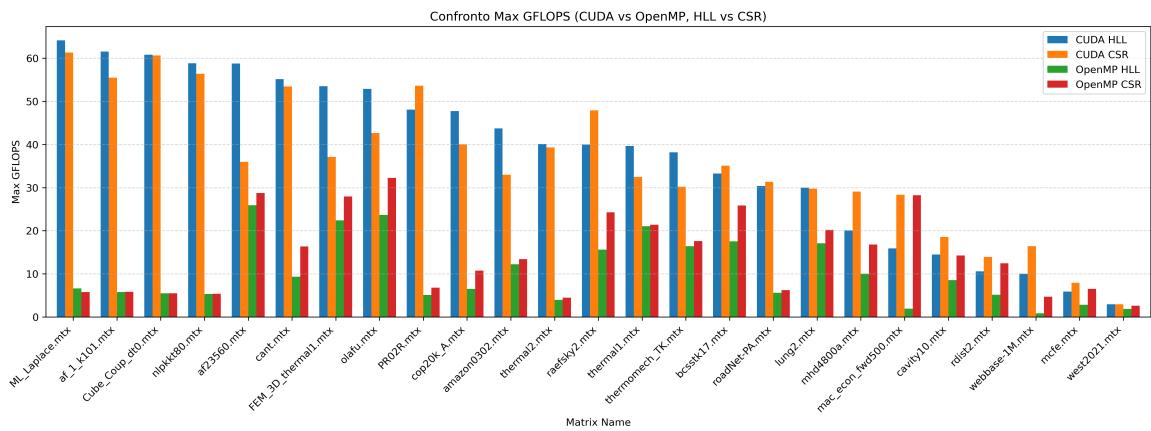


Figura 3.10: CUDA vs OPENMP

### Analisi Comparativa dei Kernel HLL

Il grafico seguente mostra le prestazioni massime in GFLOPS ottenute per ciascuna matrice, confrontando i tre kernel sviluppati per il formato *Hierarchical-ELL* (HLL): **kernel1**, **kernel2** e **kernel warp colonne**.

- Il **kernel warp colonne** si conferma il più efficiente nella maggior parte dei casi, con prestazioni significativamente superiori nelle matrici complesse o con elevata occupazione. Questo è attribuibile alla capacità del kernel di sfruttare efficacemente il parallelismo intra-warp a livello di colonne.
- I kernel **kernel1** e **kernel2**, basati su un approccio *1-thread-per-riga*, mostrano prestazioni più stabili ma mediamente inferiori. Le differenze tra i due sono minime, con leggere variazioni dovute a differenti strategie di accesso alla memoria.

- Su matrici strutturate o regolari (es. `ML_Laplace.mtx`, `af_1_k101.mtx`), le differenze tra i kernel si riducono, mentre diventano più marcate su matrici irregolari o di grandi dimensioni (es. `mhd4800a.mtx`, `bcsstk17.mtx`).
- In alcuni casi (es. `dc1.mtx`, `cage4.mtx`) le performance sono basse per tutti i kernel, suggerendo che la struttura della matrice è poco adatta al formato ELL o presenta bassa densità utile.

**Conclusione:** Il kernel *warp colonne* risulta essere il più promettente per l'elaborazione di matrici sparse in formato HLL, grazie alla sua capacità di adattarsi a strutture complesse e sfruttare al massimo l'architettura GPU. Tuttavia, i kernel alternativi offrono una soluzione più stabile in contesti meno variabili.

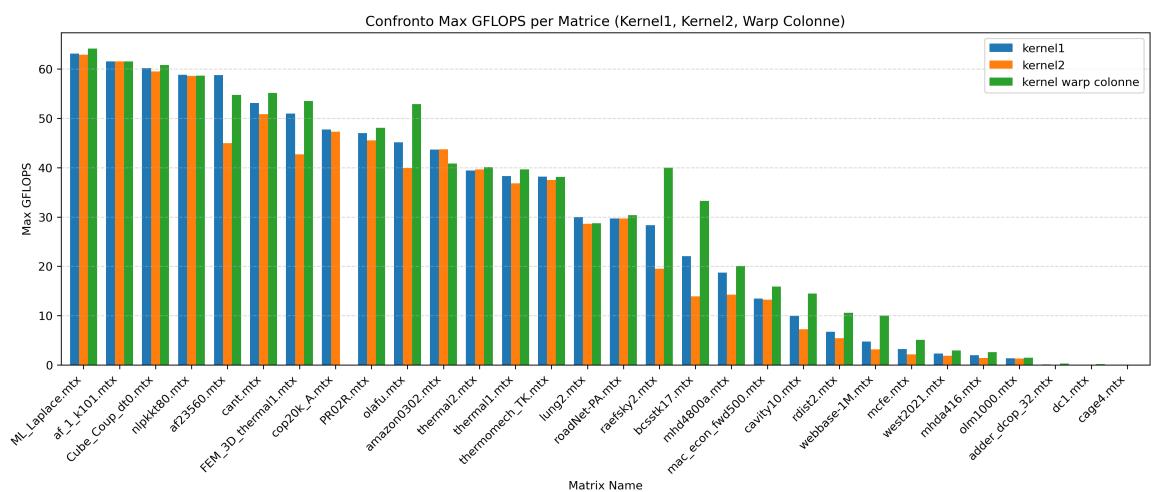


Figura 3.11: Kernel Hll

## Confronto Kernel CSR

Le differenze prestazionali osservate tra i tre kernel CUDA implementati possono essere spiegate analizzando le diverse strategie di parallelizzazione e gestione della memoria adottate.

**Confronto Kernel 1 (Row-per-Thread) vs Kernel 2/3 (Warp-per-Row):** Il primo kernel, assegnando una riga per thread, è concettualmente molto semplice e presenta un basso overhead gestionale. Inoltre, richiede un numero di thread attivi potenzialmente inferiore rispetto agli approcci warp-per-row, che ne utilizzano 32 per ogni riga processata. Questo kernel tende a comportarsi meglio degli altri due nel caso di matrici dense. Le ragioni ipotizzate, sebbene non completamente confermate dall'analisi del profiler, sono legate ai pattern di accesso alla memoria:

- Potrebbe verificarsi un miglior sfruttamento della cache per il vettore di input  $\mathbf{x}$ . Processando un'intera riga densa, un singolo thread accede a molti elementi di  $\mathbf{x}$ ; questo pattern sequenziale (dal punto di vista del thread) potrebbe favorire la località temporale o spaziale nella cache L1/L2 di quel thread/core.

- Gli approcci warp-per-row, pur avendo accessi più coalescenti ai dati della matrice (`valori`, `jValori`), potrebbero generare un pattern di accesso al vettore `x` (tramite gli indici di colonna letti con stride 32) meno favorevole per la cache rispetto all'accesso sequenziale (logico) del primo kernel su righe dense.

In sintesi, per matrici dense, il minor overhead e un potenziale (ma non verificato) miglior utilizzo della cache nel kernel row-per-thread sembrano prevalere sui benefici della parallelizzazione intra-riga dei kernel warp-per-row.

**Confronto Kernel 2 (Warp-per-Row) vs Kernel 3 (Warp-per-Row con Padding):** La differenza chiave tra questi due kernel risiede nell'impatto sulla cache L2 e sull'efficienza degli accessi alla memoria globale, dovuto all'introduzione del padding nel Kernel 3.

- **Kernel 2:** Quando un warp legge un segmento di memoria (es. 128 byte) per la riga corrente, specialmente se le righe sono corte (matrici sparse), è probabile che vengano caricati anche dati appartenenti alle righe immediatamente successive. Questo "prefetching" accidentale può aumentare significativamente il numero di *cache hit* quando quelle righe successive verranno effettivamente processate, mascherando parzialmente l'inefficienza di accessi non perfettamente coalescenti all'interno della riga corrente.
- **Kernel 3:** L'aggiunta di padding mira a ottimizzare la coalescenza per la riga corrente, assicurando che gli accessi dei 32 thread del warp siano raggruppati nel minor numero di transazioni di memoria (e infatti il memory throughput sale al 95%). Tuttavia, il padding interrompe la contiguità dei dati tra righe diverse. Di conseguenza, caricando un segmento di memoria, si carichano dati della riga corrente e padding inutile, perdendo l'effetto benefico del prefetching accidentale sulla cache per le righe successive. Questo può ridurre il tasso complessivo di cache hit.

Il Kernel 3 mostra quindi performance migliori del Kernel 2 solo in scenari specifici: quando le righe hanno una lunghezza (numero di non-zeri) tale che i benefici della coalescenza quasi perfetta ottenuta grazie al padding (specialmente se la lunghezza è vicina a un multiplo di 32 o della dimensione del segmento di memoria) superano gli svantaggi dovuti alla riduzione dei cache hit inter-riga.

**Confronto fra Shuffle e Riduzione Diretta** Per quanto riguarda i kernel che implementano la *shuffle* e la riduzione custom, ho testato tutte le configurazioni possibili compatibili con un blocco di 1024 threads. In particolare, ho variato il parametro `miniWarp` (ossia il numero di thread assegnati alle singole righe della matrice) da 2 fino a 32, come già osservato nei kernel precedenti.

Inoltre, per ciascun valore di `miniWarp`, ho assegnato a ogni blocco un numero di threads pari a un multiplo di  $32 \times 2^i$ , fino a raggiungere 1024 threads. Come si può osservare nel grafico seguente, per matrici con un numero di elementi non nulli inferiore a circa  $10^7$ , la varianza fra i risultati ottenuti dalle diverse configurazioni è piuttosto bassa, producendo quindi prestazioni simili con alcune eccezioni.

Al contrario, nel caso di matrici con un numero elevato di elementi non nulli, si osservano differenze significative tra le varie configurazioni, con scostamenti che possono raggiungere anche i 20–30 GB tra una configurazione e l'altra.

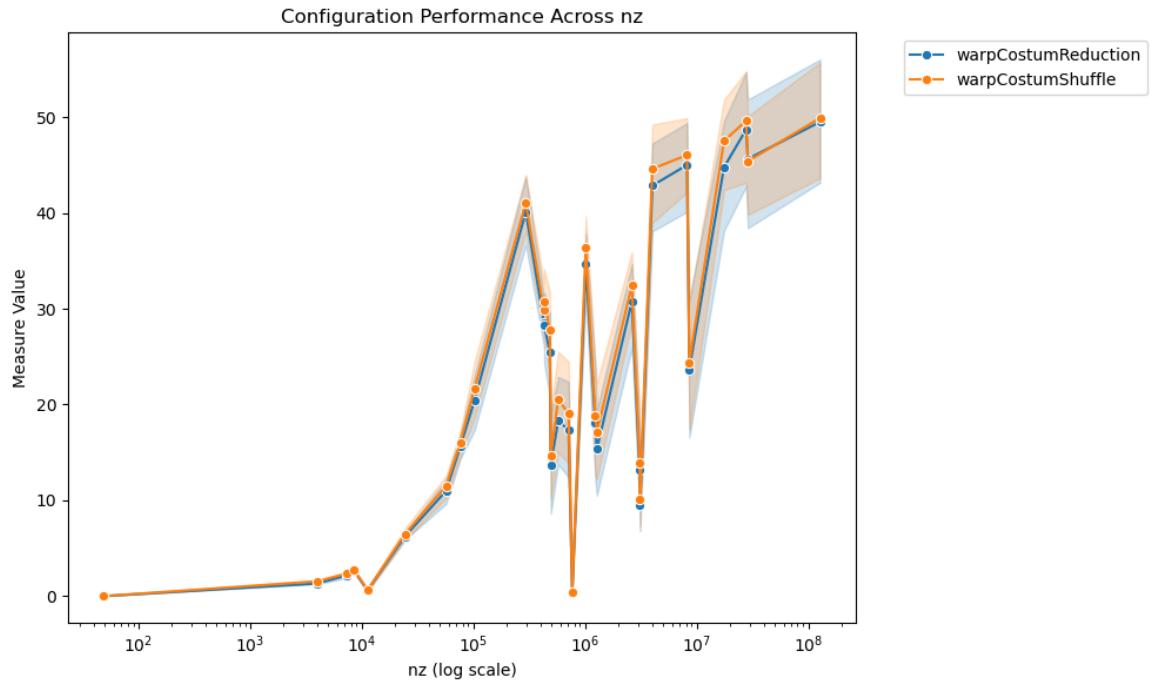


Figura 3.12: reduction vs shuffle

Per quanto riguarda le configurazioni migliori e peggiori, esse dipendono fortemente dalla struttura delle matrici considerate. In particolare, la configurazione con 4 threads per riga tende a comportarsi bene nel caso di matrici con pochissimi valori per riga, ma mostra prestazioni buone solo in questi casi specifici.

Un aspetto che ho osservato è che, aumentando il numero totale di threads all'interno del blocco, le performance tendono generalmente a degradarsi, soprattutto nel caso della riduzione.

I seguenti due grafici mostrano, rispettivamente, il caso migliore e il caso peggiore per ciascuna matrice. In ogni colonna dei grafici è indicata la configurazione sotto forma di `miniWarpSize, ThreadsPerBlock`.

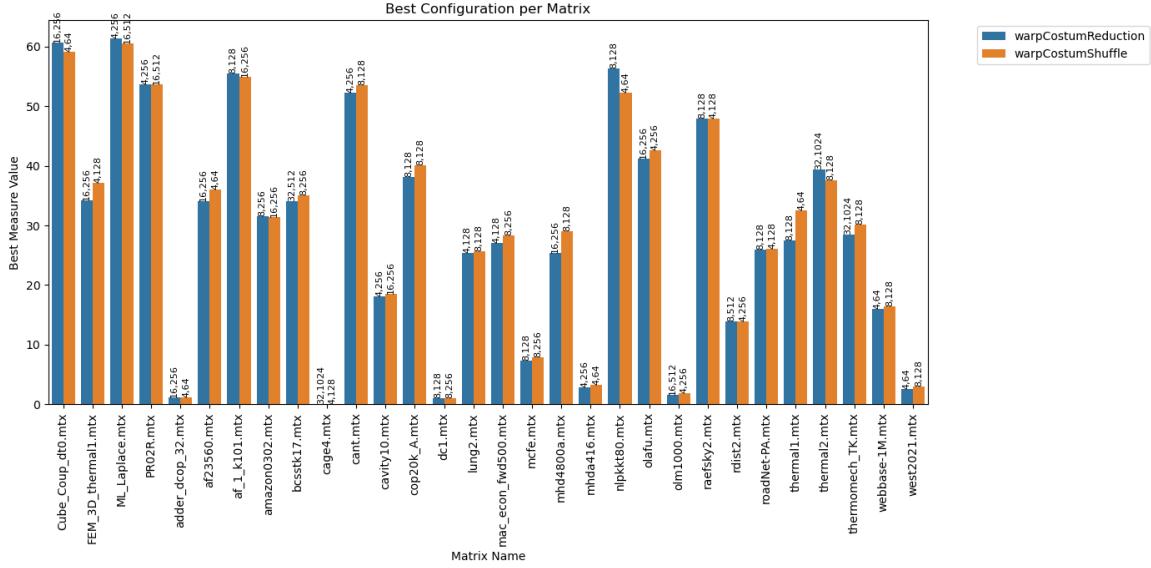


Figura 3.13: reduction vs shuffle Best

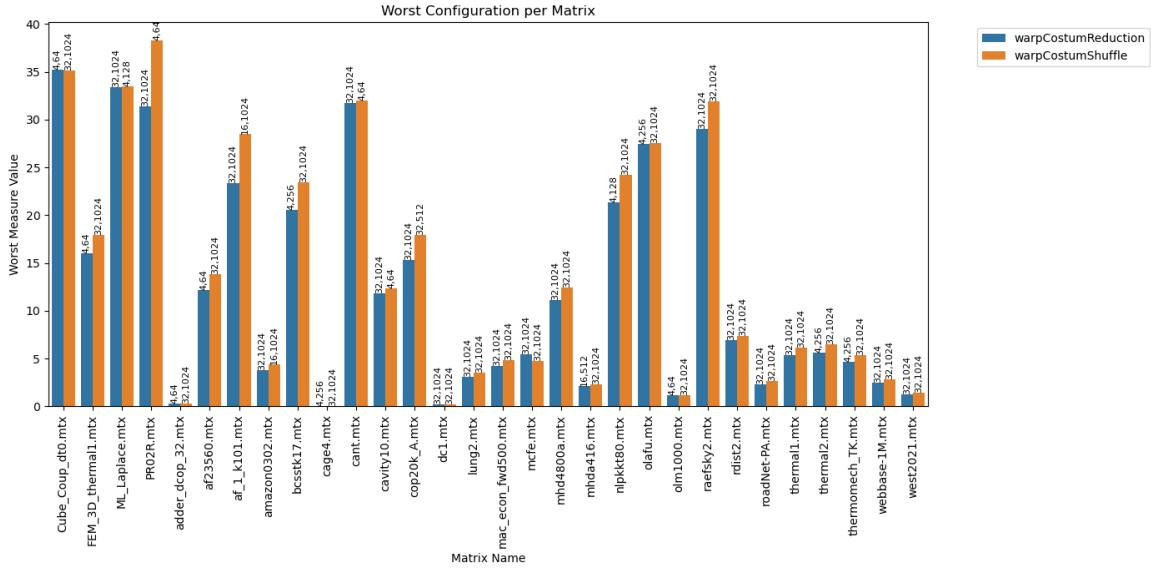


Figura 3.14: reduction vs shuffle worst

Infine, presento un grafico che rappresenta le performance di tutti i kernel CSR implementati, inclusi quelli basati su CUDA, OpenMP e l'implementazione seriale.

Le migliori performance vengono ottenute dai kernel CUDA, in particolare dai kernel `warpCustomReduction` e `warpCustomShuffle`. Il kernel `warp`, ossia l'implementazione semplice basata su warp, e una versione ottimizzata di `warpCustomShuffle` con `miniWarp` pari a 32 mostrano performance inferiori rispetto ai primi due in tutti i test.

Questo comportamento è dovuto principalmente al fatto che quasi tutte le matrici utilizzate nei benchmark sono sparse con meno di 32 elementi per riga. Di conseguenza, l'utilizzo

di **miniWarp** pari a 32 comporta un calo significativo delle performance rispetto alle configurazioni con **miniWarp** pari a 4, 8 o 16, che riescono invece a sfruttare meglio la computazione e la memoria, pur introducendo un overhead maggiore, ampiamente giustificato dai risultati ottenuti.

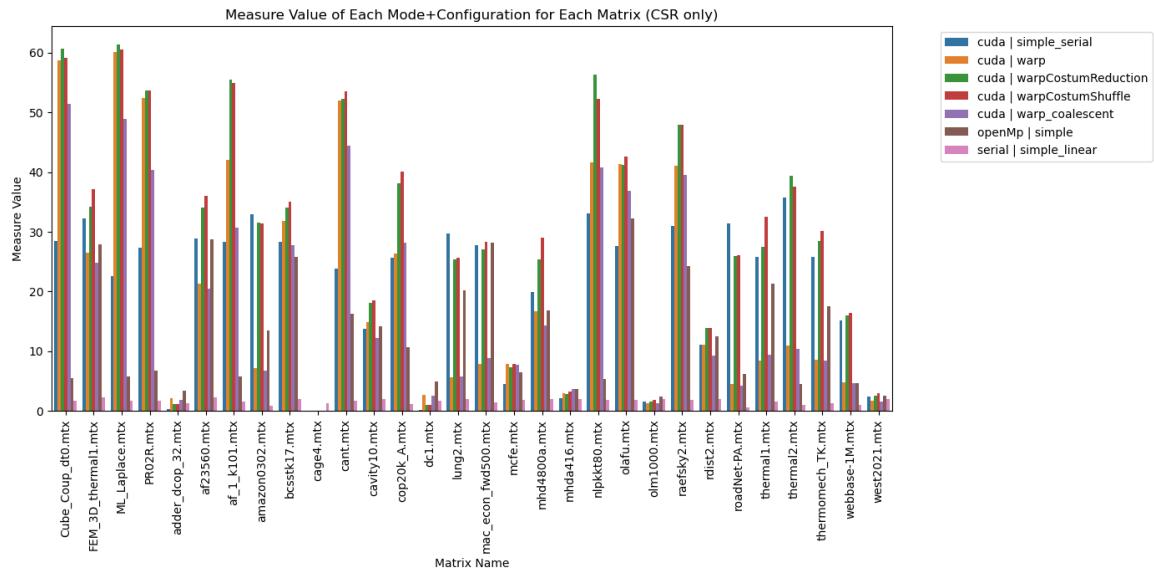


Figura 3.15: csr all

### 3.6 Dati tabellari

Max GFLOPS & Best Threads - Open MP CSR			Max GFLOPS & Best Threads - Open MP HLL		
Matrix Name	Max GFLOPS	Best Threads	Matrix Name	Max GFLOPS	Best Threads
olafu.mtx	32.242301	32	af23560.mtx	25.896007	32
af23560.mtx	28.722638	32	olafu.mtx	23.646639	32
mac_econ_fwd500.mtx	28.231149	32	FEM_3D_thermal1.mtx	22.395097	32
FEM_3D_thermal1.mtx	27.964421	32	thermal1.mtx	21.015999	32
bcsstk17.mtx	25.87617	32	bcsstk17.mtx	17.523012	32
raefsky2.mtx	24.259228	32	lung2.mtx	17.102233	32
thermal1.mtx	21.39506	32	thermomech_TK.mtx	16.391264	32
lung2.mtx	20.202703	32	raefsky2.mtx	15.597613	32
thermomech_TK.mtx	17.573594	32	amazon0302.mtx	12.224806	32
mhd4800a.mtx	16.79551	32	mhd4800a.mtx	10.007497	32
cant.mtx	16.352982	32	cant.mtx	9.3240792	32
cavity10.mtx	14.238313	16	cavity10.mtx	8.567385	16
amazon0302.mtx	13.431176	32	ML_Laplace.mtx	6.6403039	8
rdist2.mtx	12.473458	16	cop20k_A.mtx	6.5241391	32
cop20k_A.mtx	10.744792	64	af_1_k101.mtx	5.7845637	16
PR02R.mtx	6.7752949	8	roadNet-PA.mtx	5.5803494	32
mcf.e.mtx	6.5303001	16	Cube_Coup_dt0.mtx	5.481466	8
roadNet-PA.mtx	6.2454628	32	nlpkkt80.mtx	5.3047332	8
af_1_k101.mtx	5.8370468	32	rdist2.mtx	5.1701976	16
ML_Laplace.mtx	5.7576708	32	PR02R.mtx	5.072367	8
Cube_Coup_dt0.mtx	5.470798	16	thermal2.mtx	3.9835853	32
nlpkkt80.mtx	5.3690134	8	mcf.e.mtx	2.8026949	4
dc1.mtx	4.999462	32	mhda416.mtx	2.0654634	4
webbase-1M.mtx	4.6709361	64	mac_econ_fwd500.mtx	1.9182081	32
thermal2.mtx	4.4839342	32	olm1000.mtx	1.8679462	2
mhda416.mtx	3.63806	8	west2021.mtx	1.8642795	8
adder_dcop_32.mtx	3.3692245	8	webbase-1M.mtx	0.83985235	64
west2021.mtx	2.6221267	8	adder_dcop_32.mtx	0.61389428	8
olm1000.mtx	2.3810612	2	dc1.mtx	0.19634409	64
cage4.mtx	0.10256014	2	cage4.mtx	0.14454217	2

(a) OPENMP CSR

(b) OPENMP HLL

Figura 3.16: Confronto OPENMP

Max GFLOPS & Best Threads - CUDA CSR			Max GFLOPS & Best Threads - CUDA HLL		
Matrix Name	Max GFLOPS	Best Threads	Matrix Name	Max GFLOPS	Best Threads
ML_Laplace.mtx	61.330474	16	ML_Laplace.mtx	64.15894	32
Cube_Coup_dt0.mtx	60.664949	16	af_1_k101.mtx	61.572676	128
nlpkkt80.mtx	56.384501	32	Cube_Coup_dt0.mtx	60.818555	64
af_1_k101.mtx	55.481119	16	nlpkkt80.mtx	58.846133	64
PR02R.mtx	53.618171	16	af23560.mtx	58.76893	32
cant mtx	53.483117	32	cant mtx	55.131288	32
raefsky2.mtx	47.896484	16	FEM_3D_thermal1.mtx	53.521373	32
olafu.mtx	42.639279	16	olafu.mtx	52.872706	64
cop20k_A.mtx	40.044114	16	PR02R.mtx	48.09354	32
thermal2.mtx	39.341909	32	cop20k_A.mtx	47.749835	64
FEM_3D_thermal1.mtx	37.081612	64	amazon0302.mtx	43.703179	128
af23560.mtx	35.988111	64	thermal2.mtx	40.121919	32
bcsstk17.mtx	35.066263	16	raefsky2.mtx	39.983151	128
amazon0302.mtx	32.954659	32	thermal1.mtx	39.628726	128
thermal1.mtx	32.46259	64	thermomech_TK.mtx	38.206507	32
roadNet-PA.mtx	31.344487	128	bcsstk17.mtx	33.238988	64
thermomech_TK.mtx	30.212211	16	roadNet-PA.mtx	30.357103	128
lung2.mtx	29.744204	32	lung2.mtx	30.005117	64
mhd4800a.mtx	29.048863	16	mhd4800a.mtx	20.096698	64
mac_econ_fwd500.mtx	28.332791	32	mac_econ_fwd500.mtx	15.914179	64
cavity10.mtx	18.57174	64	cavity10.mtx	14.463447	32
webbase-1M.mtx	16.391859	16	rdist2.mtx	10.558976	64
rdist2.mtx	13.899903	32	webbase-1M.mtx	10.009592	32
mcfe.mtx	7.9368489	32	mcfe.mtx	5.8836874	64
mhda416.mtx	3.6905172	64	west2021.mtx	2.9459136	64
west2021.mtx	2.9649194	16	mhda416.mtx	2.7871093	32
dcl.mtx	2.7212674	32	dcl.mtx	1.7452361	32
adder_dcop_32.mtx	2.1234895	96	adder_dcop_32.mtx	1.5689174	128
olm1000.mtx	1.8229927	32	olm1000.mtx	1.4866072	128
cage4.mtx	0.015950521	16	cage4.mtx	0.022685186	128

(a) CUDA CSR

(b) CUDA HLL

Figura 3.17: Confronto CUDA

# Capitolo 4

## Conclusioni

L'esperienza maturata durante lo sviluppo di questo progetto ha permesso di comprendere più a fondo l'importanza e, al contempo, la difficoltà intrinseca che caratterizzano le attività di ottimizzazione e parallelizzazione di problemi computazionali.

È emerso chiaramente come la *scelta della rappresentazione dei dati* sia un fattore critico: decisioni come l'adozione del formato CSR piuttosto che HLL influenzano non solo la struttura del codice ma anche il potenziale di performance raggiungibile. Allo stesso modo, la selezione dei *costrutti di parallelizzazione* specifici (all'interno di OpenMP o CUDA) deve essere ponderata in base all'algoritmo e all'architettura sottostante.

Questo lavoro ha inoltre sottolineato l'importanza fondamentale dei *tool di analisi e profiling* che devono necessariamente affiancare il programmatore moderno. Strumenti come **NSight Compute**, o altri profiler, offrono uno sguardo prezioso sul comportamento effettivo del codice parallelo, permettendo di quantificare il grado di parallelizzazione, identificare colli di bottiglia e, talvolta, ricevere suggerimenti diretti per il miglioramento.

Tuttavia, è cruciale riconoscere che tali strumenti sono un ausilio, non una panacea. Non possono sostituire le *buone pratiche di programmazione* e un'attenta fase di *progettazione e ragionamento* da parte dello sviluppatore. Infatti, durante il progetto, è capitato che alcuni kernel, a causa di una loro costruzione errata o inefficiente alla radice, si siano rivelati intrinsecamente "insalvabili" o molto difficili da ottimizzare significativamente, anche con l'aiuto dei profiler.

Infine, desideriamo concludere ribadendo una riflessione comparativa sui due formati di matrice esplorati:

- Il formato **CSR** si è dimostrato concettualmente semplice, portando alla costruzione di algoritmi relativamente lineari. Questo, se da un lato semplifica l'implementazione, dall'altro offre un margine di miglioramento prestazionale intrinsecamente più limitato, specialmente per matrici con pattern di sparsità non perfettamente regolari.
- Il formato **HLL**, al contrario, presenta una struttura dati decisamente più complessa. Richiede maggiore attenzione e cura sia nella fase di conversione della matrice (inizializzazione) sia nella successiva implementazione degli algoritmi di calcolo. Tuttavia, questa complessità è finalizzata a migliorare la località e il bilanciamento del carico, offrendo il potenziale per ottenere prestazioni superiori, in particolare quando l'overhead gestionale del formato diventa trascurabile rispetto alla dimensione e alla complessità del problema risolto.

In sintesi, l'ottimizzazione e la parallelizzazione richiedono un approccio olistico che consideri attentamente dati, algoritmi, architetture e strumenti, senza mai prescindere da una solida progettazione software.

## 4.1 Suddivisione lavoro

Il lavoro sul progetto è stato suddiviso in modo equilibrato tra i membri del gruppo, Alexandru e Luca. La strategia principale è stata quella di assegnare a ciascun membro la responsabilità completa per uno specifico formato di memorizzazione delle matrici sparse:

- **Luca:** Ha preso in carico il formato **HLL**. Questo ha incluso lo sviluppo di tutte le funzioni di supporto necessarie per la gestione di tale formato, nonché l'implementazione e l'ottimizzazione delle operazioni di moltiplicazione matrice-vettore, sia nella versione seriale che nelle versioni parallelizzate con OpenMP e CUDA.
- **Alexandru:** Si è focalizzato sul formato **CSR**. Analogamente, ha sviluppato le funzioni di supporto per il formato CSR e le relative implementazioni seriali e parallele (OpenMP, CUDA).

Durante l'intero processo di sviluppo, sono state effettuate regolarmente sessioni di confronto ("tavola rotonda") per permettere a entrambi i membri di presentare i propri progressi, discutere le sfide incontrate e collaborare alla risoluzione di eventuali problemi o dubbi.

Per quanto riguarda le attività complementari:

- **Luca :** Ha curato la parte relativa all'analisi dei dati, sviluppando gli script in **Python** per l'elaborazione dei risultati dei benchmark, il calcolo delle metriche di performance e la generazione dei grafici.
- **Alexandru :** Si è occupato della creazione dell'infrastruttura di testing, realizzando l'eseguibile principale (tramite CMake e Make) che permette di lanciare i test su tutte le configurazioni desiderate (combinazioni di formati, versioni, parametri) e di salvare sistematicamente i risultati ottenuti in file formato **csv** per la successiva analisi.

Per il resto il contributo complessivo al progetto è equamente distribuito (approssimativamente 50/50), come peraltro tracciabile attraverso la cronologia dei commit sul repository **GitHub** del progetto.